

FIRST EDITION

<packt>



Advanced Smart Home Automation Solutions with Home Assistant

Elevate your home automation system with AI, Zigbee, MQTT, ESPHome, and Golioth

Marco Carvalho

Foreword by Marcelo Cabral, Senior Engineering Manager, Tubi

Advanced Smart Home Automation Solutions with Home Assistant

Elevate your home automation system with AI, Zigbee, MQTT, ESPHome, and Golioth

Marco Carvalho

<packt>

Advanced Smart Home Automation Solutions with Home Assistant

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Rohit Rajkumar

Relationship Lead: Nivedita Pandey

Project Manager: Sandip Tadge

Content Engineer: Rohan Datta

Technical Editor: Tejas Vijay Mhasvekar

Indexer: Tejal Soni

Production Designer: Aparna Bhagat

Growth Lead: Namita Velgekar

Production reference: 1250926

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80638-469-3

www.packtpub.com

To God, for His guidance, and to my wife and my sons, for their love and support.

– Marco Carvalho

Foreword

I started my own Home Assistant setup entirely by myself, which is a polite way of saying I spent several nights arguing with YAML files and mostly losing. There was no mentor, no forum post that quite matched my situation, just me, a Raspberry Pi, and a stubborn refusal to admit defeat to a light switch.

Marco and I had been friends for a while, but somehow home automation had never once come up between us. Proof, I suppose, that even good friends run out of small talk eventually. Then one day he mentioned, almost in passing, that he'd just published a book on Home Assistant. I said, "No way!" and told him that I'd been quietly automating my own house for months. Neither of us saw that coming.

I picked up his book expecting a polite skim. Instead, I spent that weekend discovering features I hadn't even known existed, and muttering "Why didn't I think of that?" more times than I'd like to admit. That first book took the beautiful obsession of home automation and made it usable for the rest of us. Many of you built your first sensor because of it. Some of you, I suspect, are still finding stray ESP8266 boards in kitchen drawers.

Now Marco is back, and he has clearly decided that "the lights turn on automatically" was just the warm-up. This time, we're doing VLANs. We're doing Zigbee mesh networks. We're teaching Home Assistant to notice your habits and start optimizing your life before you've had coffee. If the first book taught your house to listen, this one teaches it to think, and possibly to judge you a little.

What excites me most about this book isn't just the technology; it's that Marco writes advanced material the way he explains things over dinner: patiently, practically, and with the quiet assumption that you're perfectly capable of building it yourself; you just haven't yet.

So here is what to expect. Your YAML will get more elegant and slightly more intimidating. Your network diagram will start looking like something out of a spy film. Your smart home will go from "responsive" to "borderline opinionated." And somewhere around the Golioth chapter, you will catch yourself explaining cloud IoT architecture to a family member who only wanted to know why the porch light turns purple sometimes.

Congratulations, and welcome to the advanced class. Keep a spare SD card handy, be patient with your firmware flashes, and enjoy the ride!

Marcelo Cabral

Senior Engineering Manager, Tubi

Contributors

About the author

Marco Carvalho is an electronics engineer with a master's degree in computer science and over 25 years of experience in product development, embedded systems, hardware-software integration, NPI, sustaining engineering, and engineering management across the medical, mining, and technology industries. Passionate about IoT and home automation, Marco enjoys developing custom hardware and software solutions using sensors, networking, embedded technologies, and Home Assistant as a personal interest. He is also the author of *Building Smart Home Automation Solutions with Home Assistant*.

I would first like to thank God for giving me the strength, perseverance, and opportunity to complete this book.

I am deeply grateful to my wife and sons for their love, patience, and understanding during the many evenings and weekends dedicated to writing, testing, and developing the projects presented in this book.

I would also like to thank my friends who supported and encouraged me throughout the development of this second book. Their interest, feedback, and encouragement helped motivate me during the writing process.

I would also like to thank the leadership and colleagues at my workplace for their support and for providing a professional environment that has continuously encouraged learning, engineering development, and the exploration of new technologies.

My sincere thanks also go to the technical reviewers and to the entire Packt team for their guidance, valuable feedback, and support throughout the development of this book.

About the reviewer

Taher Ujjainwala is an Electrical and Embedded Systems Engineer at Milwaukee Tool with eight years of experience in embedded systems and systems engineering. His expertise includes connectivity architectures, MBSE, hardware and firmware development, Bluetooth Low Energy, low-power systems, and connected products. He holds a Master of Engineering from the University of Colorado Boulder and is an INCOSE Associate Systems Engineering Professional (ASEP). As a technical reviewer, he brings practical industry experience and systems thinking to bridge the gap between theory and real-world application.

Table of Contents

Preface	xxi
Part 1: Home Assistant Advanced Features and Configuration	1
Chapter 1: Home Assistant Architecture and Performance Optimization	3
Technical requirements.....	4
Home Assistant behind the scenes: core components and interactions.....	5
Home Assistant operating system (HAOS) • 7	
Supervisor • 8	
Apps • 8	
Core • 9	
Integrations • 10	
<i>Types of integrations in Home Assistant • 10</i>	
<i>Developing an integration • 11</i>	
<i>Custom integration development • 12</i>	
<i>Turning a custom integration to a core integration • 17</i>	
Frontend • 18	
Understanding data flow and state management.....	18
Performance profiling and optimization techniques.....	21
Database optimization • 21	
<i>Identifying noisy entities and events • 23</i>	
<i>Database maintenance: filtering undesired information • 25</i>	
Logging configuration • 27	
Integration minimalism • 28	
Resource monitoring • 28	
Practical example: diagnosing a slow system	29
Summary	30
	31

Chapter 2: Mastering YAML for Complex Home Assistant Configurations

Technical requirements.....	32
Advanced YAML syntax: anchors, aliases, and includes.....	32
Anchors and aliases • 33	
Includes • 34	
Directive <i>!include file.yaml</i> • 36	
Directive <i>!include_dir_list folder/</i> • 37	
Directive <i>!include_dir_merge_list folder/</i> • 38	
Directive <i>!include_dir_named folder/</i> • 38	
Directive <i>!include_dir_merge_named folder/</i> • 39	
Modularizing your configuration: strategies for scalability	40
Strategy A — breaking configuration into domain-specific files • 41	
Strategy B — organizing by function • 42	
Best practices on folder structure and naming conventions • 43	
Version control with Git: managing your Home Assistant configuration	44
Create a backup routine using GitHub • 45	
Retrieve backup files using GitHub • 47	
<i>Shell command and shell script development</i> • 48	
<i>Helper development</i> • 50	
<i>Script development</i> • 51	
<i>Dashboard integration</i> • 54	
Secrets management and sensitive data handling.....	55
Using <i>secrets.yaml</i> in Home Assistant • 55	
Troubleshooting YAML: common pitfalls and debugging techniques	56
Understanding YAML parsing in Home Assistant • 56	
Common YAML errors and how to fix them • 57	
Using built-in debugging tools • 58	
Preventive best practices • 59	
Summary	59

Chapter 3: Advanced Templating in Home Assistant

Technical requirements.....	62
Jinja2 fundamentals in Home Assistant: variables, filters, and tests.....	62
Understanding Jinja2 syntax and integration with Home Assistant • 63	

Working with variables • 64	
Using filters • 64	
Using tests • 66	
Dynamic data extraction and manipulation.....	67
Getting data: states, attributes, and groups • 68	
Selecting and transforming entity lists • 69	
Numeric calculations (sum, min, max) • 70	
Advanced templating: conditionals, loops and complex logic	72
Conditionals: indirect implementation and if, elif, else • 72	
<i>Conditional - indirect implementation • 72</i>	
<i>Nested if/elif/else • 74</i>	
Loops: using for ... in • 75	
Using the Home Assistant Template Editor for debugging and testing.....	78
Common template errors and fixes • 79	
Real-world application: dynamic notifications, sensors, and UI customization.....	81
Energy price calculation • 83	
Refrigerator energy and cost • 84	
Threshold and notification • 86	
Dashboard • 88	
Summary	91
 Part 2: Communication Protocols and Cloud Integration	 93

Chapter 4: Securing Your Smart Home Network	95
--	-----------

Technical requirements.....	96
Understanding smart home vulnerabilities and threat landscape.....	96
Common vulnerabilities in IoT and smart home systems • 97	
Typical threat actors • 98	
Case studies: real-world breaches • 99	
Defense in depth: a layered security approach • 100	
Network segmentation: VLANs and firewall rules for IoT isolation.....	101
VLAN and subnet fundamentals • 101	
Designing and implementing a segmented smart home network • 102	
<i>Choosing your router • 103</i>	
<i>Setting up the VLANs and creating the interfaces • 104</i>	

<i>Configuring firewall and traffic routes</i>	• 106
<i>Enabling IoT wireless interface</i>	• 109
Secure remote access: VPN, Nabu Casa, and Cloudflare Tunnel	110
Risks of direct port forwarding	• 111
VPN-based remote access (WireGuard or OpenVPN)	• 111
<i>VPN - server mode</i>	• 111
<i>VPN – client mode</i>	• 112
Nabu Casa remote access	• 113
Cloudflare Tunnel: free, secure, and powerful	• 113
Managing SSL certificates with Let's Encrypt and DuckDNS	• 113
Device hardening: firmware, passwords, and service management	114
The importance of firmware updates	• 114
Dealing with passwords	• 115
Disabling unused services (Telnet, SSH, Web UI)	• 115
Secure OTA updates and device renaming	• 116
Monitoring new devices connecting to the network	• 116
Security monitoring, auditing, and incident response	117
Using Home Assistant and Apps for log and event monitoring	• 117
Detecting suspicious behavior	• 118
Building alerts with automations and notifications	• 118
Creating a simple incident response workflow	• 119
Summary	120
Checklist – device hardening quick steps	121
 Chapter 5: Building Robust Zigbee Networks	 123
Technical requirements	124
Zigbee protocol fundamentals: device types, mesh, and channels	124
Device Types	• 125
<i>Zigbee coordinator</i>	• 126
<i>Zigbee router</i>	• 126
<i>Zigbee end device</i>	• 127
Mesh network	• 127
Communication channels	• 128
Zigbee2MQTT: advanced configuration and device management	130
Z2M installation and firmware flashing	• 130

Network configuration and pairing devices • 131	
Binding, link quality metrics, OTA and groups • 132	
Zigbee Home Automation (ZHA): setup and optimization	133
Zigbee Home Automation installation and device pairing • 133	
Zigbee Home Automation optimization • 135	
Comparison between ZHA and Z2M: when to use one or the other • 135	
Building a robust mesh: router placement and network health	136
Coordinator placement best practices • 137	
Router selection: choosing the right Zigbee repeaters • 137	
Determining the right number of routers for a Zigbee network • 137	
Diagnosing Zigbee signals with LQI and RSSI • 138	
Identifying poor-quality devices and evaluating mesh improvements • 139	
Hands-on project: integrating Wi-Fi and Zigbee for garage gate control.....	141
Zigbee garage gate control overview • 141	
Mechanics and fast prototyping • 143	
Electronics board • 146	
Zigbee garage motor automation • 147	
Summary	151

Chapter 6: Cloud IoT Integration with Golioth 153

Technical requirements.....	154
Cloud IoT platforms: benefits and use cases.....	154
Benefits of cloud connectivity in a smart home • 154	
Common use cases for cloud integration • 155	
<i>Devices outside your home network • 156</i>	
<i>Mobile or roaming devices • 156</i>	
<i>Long-term data retention • 156</i>	
<i>Fleet management • 156</i>	
<i>Cloud-based data collection for machine learning and AI • 156</i>	
Golioth architecture and components.....	157
Core Golioth components • 157	
<i>Device identity • 158</i>	
<i>Secure communication • 158</i>	
<i>Telemetry streams • 159</i>	
<i>Settings service • 159</i>	

<i>OTA firmware updates</i> • 159	
<i>Web console and APIs</i> • 159	
<i>Embedded devices</i> • 159	
Connecting ESP32 devices to Golioth	160
ESP32-based temperature sensor circuit • 160	
Firmware architecture overview • 161	
Connect a device to the Golioth cloud • 162	
Integrating Golioth data with Home Assistant	165
Configuring the temperature sensor to output Golioth states • 166	
Adding OTA feature to the temperature sensor • 167	
Pull data from Golioth cloud to Home Assistant • 169	
Exploring other cloud IoT platforms	171
AWS IoT Core • 171	
Microsoft Azure IoT Hub • 171	
Particle Cloud • 172	
Arduino Cloud • 172	
Summary	174
 Part 3: Custom Device Development with ESPHome and MQTT	 177
Chapter 7: Creating Custom Devices with ESPHome	179
Technical requirements	180
Introducing ESPHome	181
Understanding ESPHome's role in Home Assistant systems • 181	
Exploring supported hardware platforms, peripherals, and communication buses • 182	
Comparing ESPHome with traditional firmware development • 184	
Understanding the core concepts behind ESPHome • 185	
Configuring, compiling, and uploading an ESPHome device	186
Configuring the ESPHome device • 187	
Generating code for the ESPHome device • 190	
Compile and upload the code to the ESPHome device • 194	
Integrating ESPHome devices with Home Assistant using the native API	195
Using the native ESPHome API • 195	
Understanding ESPHome device discovery and entity creation • 196	
Implementing advanced ESPHome features and on-device logic	198

Using lambdas and templates • 198	
Implementing on-device automation with actions and triggers • 199	
Creating reusable scripts and global variables • 200	
Optimizing power consumption with Deep Sleep • 201	
Building a Smart Trash Collection Monitor using the MPU-6500	203
Hardware layer • 204	
ESPHome firmware • 205	
Home Assistant automation logic • 210	
Summary	211
 Chapter 8: Exploring the MQTT Ecosystem	 213
Technical requirements.....	214
MQTT Fundamentals: Protocol, Broker, Clients, Topics, and QoS	214
Understanding the MQTT communication model • 215	
Core MQTT components • 215	
Topics and topic hierarchies • 216	
MQTT Wildcards • 217	
MQTT Quality of Service (QoS) • 217	
Setting up and securing the Mosquitto MQTT broker	218
Mosquitto installation and basic configuration using the Home Assistant app • 219	
Mosquitto MQTT authentication and security • 220	
Mastering MQTT Explorer for real-time monitoring and debugging	221
MQTT Explorer overview • 222	
Inspecting payloads and message metadata • 224	
Publishing test messages • 226	
Advanced MQTTX usage: testing and simulation	227
MQTTX overview • 228	
Creating a car GPS virtual device using MQTT discovery • 229	
Testing the GPS MQTT topics and automations using MQTTX • 232	
<i>Connectivity test • 232</i>	
<i>GPS data validation • 233</i>	
<i>Garage approach detection test • 233</i>	
<i>Automation testing • 233</i>	
Hands-on project: GPS tracking and location-based automations	234
Hardware and system architecture • 235	

GPS configuration, data and sentence • 236	
MQTT GPS solution sensor software implementation • 238	
<i>MQTT topic design</i> • 238	
<i>System flow</i> • 239	
<i>Core software structure</i> • 239	
<i>Key function overview</i> • 240	
Integrating GPS data into Home Assistant and building automations • 242	
Summary • 247	

Chapter 9: Building Touchscreen Display and Camera Devices **249**

Technical requirements.....	250
Main project overview and system architecture	250
System architecture overview • 251	
Understanding display and touch technologies • 252	
Hardware interfacing: connecting displays, touchscreens, and cameras to ESP32	253
Building a TFT touchscreen software application using ESPHome and MQTT	254
Configuring the display and touchscreen in ESPHome • 256	
Configuring the MQTT for the application • 259	
Building the user interface with LVGL support • 259	
<i>Building screen layouts using ESPHome LVGL Designer</i> • 261	
<i>Implementing navigation between pages</i> • 262	
<i>Implementing MQTT publish functionality on buttons</i> • 266	
<i>Storing configuration using global variables for persistence</i> • 267	
<i>Creating status indicator icons for Wi-Fi, MQTT, and clock</i> • 268	
<i>Implement a timer feature to display brightness and screen inactivity control</i> • 269	
Integrating cameras with ESPHome and Home Assistant	271
Introducing ESP32 camera sensors • 271	
Approach and the camera sensor used for the application • 273	
Configuring a camera sensor module in ESPHome • 273	
ESP32-S3 camera as an independent device	277
Hardware architecture • 277	
Camera software features and implementation • 278	
<i>HTTP-based camera interface feature</i> • 278	
<i>Network independence with mDNS</i> • 280	
<i>Wi-Fi provisioning via captive portal</i> • 280	

Integrating the ESP32-S3 camera with Home Assistant • 281	
Summary	283
Part 4: Advanced Automation Patterns and AI Integration	285
Chapter 10: Advanced Automation and External Logic	287
Technical requirements.....	288
Beyond basic automations: state machines, sequences, and conditional flows.....	288
Thinking in systems instead of individual rules • 289	
State machines in home automation • 290	
Separating state logic from action logic • 292	
Sequences and multi-step automations • 293	
Combining state machines and conditional flows • 294	
AppDaemon: Python-based automation and custom applications	296
What is AppDaemon? • 297	
Installing AppDaemon • 298	
Basic AppDaemon application structure • 299	
Making the app reusable • 301	
HADashboard in AppDaemon • 303	
Enabling and creating a basic HADashboard • 303	
Node-RED: visual programming for complex integrations and flows.....	305
What is Node-RED? • 305	
Installing and using Node-RED • 306	
Key groups and node types • 308	
Orchestrating external automation logic: choosing the right tool.....	309
Comparing the three automation approaches • 309	
Automation tool decision guidelines • 310	
Hybrid architecture: combining the tools • 311	
Practical example: hybrid security automation using Home Assistant, Node-RED, and AppDaemon • 312	
State layer - Home Assistant helpers and groups • 313	
Flow layer – Node-RED description • 314	
Logic layer – AppDaemon processing • 316	
Testing the automation • 317	
Designing resilient home automation systems.....	318

Summary	321
 Chapter 11: Extending Home Assistant with HACS	 323
Technical requirements.....	324
Introduction to HACS: the Home Assistant Community Store.....	324
Installation, configuration, and troubleshooting HACS contents.....	326
Installing HACS content • 327	
Configuring HACS content and confirming that it works • 328	
Troubleshooting HACS content • 329	
Curated advanced HACS content: beyond the official.....	331
Recommended categories for advanced users • 331	
Example 1 – Advanced dashboard cards • 332	
Example 2 – System monitoring integrations • 333	
Example 3 – Custom device integrations • 335	
Example 4 – Themes for usability and wall-panel dashboards • 337	
Understanding custom component structure and development basics.....	337
Practical example – preparing the Dual Measurement Sensor integration for HACS • 339	
Maintaining and updating your HACS ecosystem	340
Recommended HACS update workflow • 341	
HACS maintenance checklist and what to verify after an update • 341	
Summary	342
 Chapter 12: AI-Driven Home Automation	 345
Technical requirements.....	346
AI and ML in home automation: concepts and possibilities	346
Rules, Statistics, ML and LLMs • 348	
Edge AI versus cloud AI • 349	
What AI should and should not control in your home • 351	
Practical possibilities for AI in smart homes • 352	
Learning consumption habits from Home Assistant data.....	352
Data sources and quality • 353	
Preparing sensor data to learn usage habits • 355	
Using AI Task in Home Assistant • 357	
Updating helpers from AI results • 360	
Creating shadow entities • 361	

Sending notifications from shadow sensors and running periodic AI trend analysis • 362	
Other AI Task examples • 364	
Implementing rule-based and simple ML-driven suggestions	365
Rule-based suggestion • 365	
Implementing a simple machine learning recommendation • 368	
Anomaly detection for security and efficiency • 370	
Common smart home anomalies • 370	
AI as an explanation layer • 371	
Best practices in anomaly detection • 373	
Integrating external ML services for advanced AI capabilities.....	373
Understanding AI integration options • 373	
Using MCP to connect AI assistants • 374	
AI-powered camera analysis • 379	
<i>Frigate</i> • 379	
<i>LLM Vision</i> • 379	
Summary	380
Final message.....	381
 Chapter 13: Unlock the Code Bundle and the PDF Version	 383
Unlock this book's free benefits in three easy steps	384
 Other Books You May Enjoy	 388
Index	391

Preface

Home automation continues to evolve rapidly. What was once primarily about remotely switching lights or scheduling devices has expanded into an ecosystem of connected sensors, local and cloud services, multiple communication protocols, custom hardware, advanced automation logic, and, increasingly, artificial intelligence. Home Assistant has evolved alongside these technologies and provides a flexible platform for bringing them together in a single home automation environment.

The idea for this second book came largely from feedback I received from readers of my first book, *Building Smart Home Automation Solutions with Home Assistant*. Some readers asked about technologies that were not covered, such as Zigbee, while others expected certain subjects to go further or provide a deeper and more advanced treatment. At the same time, while writing the first book, there were many technologies, tools, protocols, sensors, and project ideas that I simply did not have the opportunity to include or discuss in more depth. Together, these factors motivated me to write this second book and explore many of those subjects beyond the foundations covered previously.

After completing my first book, I also remained as motivated as ever to talk about home automation, experiment with new technologies, and find new ways of integrating hardware and software into useful systems. I am especially interested in seeing where Home Assistant, embedded devices, local processing, artificial intelligence, new wireless protocols, and other adjacent technologies will take home automation next.

This book can therefore be considered a next step after *Building Smart Home Automation Solutions with Home Assistant*, but reading the first book is not a requirement. If you already have a working Home Assistant installation and are familiar with its basic concepts, this book can also serve as a step forward into technologies or approaches that you may not have explored before.

The intention of this book is not to suggest that every subject will be advanced or completely new to you. Home automation brings together many different disciplines, including software, networking, electronics, embedded systems, communication protocols, cybersecurity, cloud services, and artificial intelligence. You may already know one of these areas very well and decide to move quickly through that material. In another chapter, however, you may encounter a technology or approach that you have never used before.

The examples and projects presented throughout the book are practical implementations. The configurations, devices, communication methods, and automations described in the projects were implemented and tested before and during the development of the book. Home Assistant and the adjacent technologies used with it continue to evolve, so interfaces, menu locations, software versions, and individual configuration options may change after publication. The underlying concepts and development approaches, however, should provide a foundation that you can adapt as these technologies evolve.

I hope this book encourages you not only to reproduce the projects presented here, but also to modify them, combine their ideas, and create solutions that address your own needs.

If you learn even one useful concept, discover one technology that encourages you to experiment, or find one project idea that improves your own home automation system, I will consider the purpose of this book accomplished.

Who this book is for

This book is intended for Home Assistant users who already understand the fundamentals of home automation and want to expand their systems beyond standard device integrations and basic automations.

Home automation and IoT enthusiasts who enjoy experimenting with custom devices, sensors, communication protocols, networking, and automation logic will find practical projects and techniques throughout the book. Electronics and embedded systems engineers can use the projects to explore the integration of ESP-based hardware, sensors, displays, cameras, MQTT, ESPHome, and cloud IoT technologies with Home Assistant.

IT and networking professionals who are interested in home automation will find material covering network segmentation, VLANs, firewall rules, secure remote access, Zigbee networks, and IoT device security. Software developers can explore YAML, Jinja templating, Python-based automation approaches, custom integrations, and AI-related applications.

The book may also be useful to product and technology professionals interested in understanding how open-source home automation, embedded devices, networking, cloud services, and artificial intelligence can be combined to create new IoT solutions.

You do not need to be an expert in all these areas. In fact, the multidisciplinary nature of the book is intentional. Readers with a software background may learn more from the hardware-oriented chapters, while electronics engineers may find the networking, templating, or automation chapters more valuable.

What this book covers

Chapter 1, Home Assistant Architecture and Performance Optimization, looks behind the Home Assistant user interface to explain its major architectural components and how they interact. You will explore Home Assistant Core, Supervisor, apps, integrations, the state machine, event bus, and service registry. The chapter also introduces integration development and practical techniques for analyzing database activity, resource usage, logging, and other potential sources of performance problems.

Chapter 2, Mastering YAML for Complex Home Assistant Configurations, expands on the use of YAML beyond simple configuration examples. You will learn techniques for structuring and maintaining larger configurations, organizing files, protecting sensitive information, using development tools, and troubleshooting configuration problems.

Chapter 3, Advanced Templating in Home Assistant, explores templating and the use of Jinja within Home Assistant. You will manipulate entity states and attributes, apply conditions, filters, and loops, use the Template Editor for testing and debugging, and build dynamic sensors, notifications, and automation logic.

Chapter 4, Securing Your Smart Home Network, examines the security risks associated with connected home devices and shows how network segmentation can reduce those risks. You will work with VLANs, firewall rules, secure remote connectivity, device hardening, and other practical measures for protecting a Home Assistant and IoT environment.

Chapter 5, Building Robust Zigbee Networks, explores Zigbee architecture, coordinators, routers, end devices, channels, mesh behavior, signal quality, and network planning. You will work with Zigbee2MQTT and ZHA and learn techniques for configuring, monitoring, troubleshooting, and improving a Zigbee network. Practical examples demonstrate how Zigbee and Wi-Fi devices can coexist within Home Assistant automations.

Chapter 6, Cloud IoT Integration with Golioth, introduces cloud IoT concepts and demonstrates how a platform such as Golioth can extend device management beyond the local network. You will explore device provisioning, data communication, remote management, over-the-air updates, and techniques for connecting cloud-managed embedded devices with Home Assistant.

Chapter 7, Creating Custom Devices with ESPHome, shows how ESPHome can be used to transform microcontroller-based hardware into custom Home Assistant devices. You will configure sensors and actuators, explore device-side processing and power-management techniques, compare relevant development approaches, and build practical sensor-based projects.

Chapter 8, Exploring the MQTT Ecosystem, explains MQTT communication through brokers, clients, topics, payloads, quality of service, and related concepts. You will configure Mosquitto and use tools such as MQTT Explorer and MQTTeX to inspect, publish, subscribe to, test, and debug MQTT traffic. A GPS-based project demonstrates how a custom device can publish location information and trigger Home Assistant automations.

Chapter 9, Building Touchscreen Display and Camera Devices, brings several of the technologies covered in previous chapters together. You will work with ESP32-based displays, touch interfaces, graphical user interfaces, cameras, ESPHome, and MQTT to create interactive devices that exchange information with Home Assistant.

Chapter 10, Advanced Automation and External Logic, moves beyond simple trigger-condition-action automations and introduces approaches for modeling more complex behavior. You will explore state machines, conditional flows, external automation logic, and Python-based applications with AppDaemon, together with considerations for selecting an appropriate automation approach.

Chapter 11, Extending Home Assistant with HACS, explores the Home Assistant Community Store and the ecosystem of community-developed integrations and frontend resources. You will learn how HACS content is installed, configured, validated, maintained, and troubleshot, while also gaining an understanding of how community extensions complement Home Assistant's official integrations.

Chapter 12, AI-Driven Home Automation, explores how artificial intelligence and machine learning concepts can be applied to information already available in a smart home. You will analyze historical Home Assistant data, identify patterns and anomalies, create rule-based and machine-learning-assisted suggestions, and explore local, edge, and external AI approaches that can make an automation system more adaptive.

To get the most out of this book

This book assumes that you already have access to a working Home Assistant installation. Installation of Home Assistant itself is not covered because the focus is on extending and improving an existing environment. If you need guidance on installing and initially configuring Home Assistant, refer to *Building Smart Home Automation Solutions with Home Assistant* or the official Home Assistant documentation.

You should be comfortable navigating the Home Assistant interface, adding integrations and devices, creating basic automations, and editing configuration files when required. Familiarity with YAML is useful, although the book develops this knowledge further in Chapters 2 and 3.

Because the projects span software, networking, and embedded hardware, no single technical background is required for every chapter. However, the following will help you get the most from the examples:

- A working Home Assistant installation, preferably Home Assistant OS for examples that use apps
- Administrator access to your Home Assistant instance
- A computer connected to the same network as Home Assistant
- Basic knowledge of Home Assistant entities, devices, integrations, automations, and dashboards
- Basic familiarity with YAML
- Basic networking knowledge, including IP addresses, routers, Wi-Fi, and local networks
- The ability to install desktop applications and development tools
- Basic command-line familiarity for some examples
- Basic programming knowledge for chapters involving Python, templates, or embedded development
- Basic electronics knowledge for projects involving ESP32 devices, sensors, displays, and other hardware
- Appropriate ESP32 or similar development boards and sensors for the hands-on projects you choose to reproduce

Some chapters require additional software or hardware, but these requirements are identified within their respective Technical Requirements sections. You therefore do not need to purchase every device discussed in the book before you begin.

You also do not need to complete every chapter sequentially. Parts 1 and 2 provide useful foundations for many of the later projects, but readers who already have experience in a particular subject can move directly to the chapters most relevant to them.

All projects should be adapted to your own environment. Entity names, IP addresses, device identifiers, GPIO assignments, network configurations, credentials, and other values shown in the examples are illustrative and will normally need to be changed for your installation.

During the development of this book, Home Assistant evolved from approximately version **2025.8 to 2026.7**. Because Home Assistant is updated frequently, some features, menu sequences, button locations, labels, integration screens, or configuration options shown in the book may have changed slightly by the time you read it. The examples and procedures reflect the versions available when each chapter was written and tested. If you notice differences in

the interface or terminology, refer to the current Home Assistant documentation while following the same underlying concepts and workflow described in the book.

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to **packtpub.com**
2. Click your **profile picture** and select **Your Orders**
3. Find this book and click **Download Code**

If you bought this book from Amazon or any other channel partner:

1. Go to packtpub.com/unlock or scan the following QR code:



2. Search for this book
3. Sign up or log in to your free Packt account
4. Upload your proof of purchase and download the code bundle locally

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to the *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "create a backup of the home-assistant_v2.db file."

A block of code is set as follows:

```
template:
  - sensor:
    - name: "Average Indoor Temperature"
      unit_of_measurement: "°C"
      state: >
        {{ ((states('sensor.living_room_temperature') | float +
              states('sensor.bedroom_temperature') | float) / 2) | round(1) }}
```

Any command-line input or output is written as follows:

```
ssh root@192.168.8.1
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Go to **Settings** | **Devices & services** | **Helpers**."

Note

Warnings or important notes appear like this.

Tip

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at customercare@packt.com and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packt.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packt.com/>.

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



Complete Code Bundle

Download the full source code for this book's examples and projects.



DRM-Free PDF Version

Download DRM-free PDF and ePub copies of this book.



7-Day Packt Library Access

Get 7-day unlimited access to 8,000+ books and videos. No credit card required.

Available for first-time Packt+ trial users only.



Next-Gen Reader Access

Read this book on Packt Reader with progress sync, dark mode and note-taking.

How to unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

Share your thoughts

Once you've read *Advanced Smart Home Automation Solutions with Home Assistant*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



<https://packt.link/r/1806384698>

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Subscribe to Inside Embedded!

We are excited to introduce **Inside Embedded**, our brand-new newsletter dedicated to everything embedded systems development. Whether you're working with firmware, embedded Linux, microcontrollers, IoT, or connected devices, every issue brings you curated insights, practical workflows, and expert analysis to help you spend less time searching and more time building smarter, more reliable systems. Sign up now and get exciting benefits.



<https://packt.link/adqzC>

Part 1

Home Assistant Advanced Features and Configuration

In this part, you'll explore how Home Assistant works behind the scenes, optimize its performance, and use advanced YAML and templating techniques. By the end of this part, you'll be able to better organize, troubleshoot, and customize complex Home Assistant configurations.

This part of the book includes the following chapters:

- *Chapter 1, Home Assistant Architecture and Performance Optimization*
- *Chapter 2, Mastering YAML for Complex Home Assistant Configurations*
- *Chapter 3, Advanced Templating in Home Assistant*

1

Home Assistant Architecture and Performance Optimization

This book is written for those who already use Home Assistant and want to move beyond standard configurations into robust, scalable, and deeply customized smart home systems. Throughout the book, you will learn how Home Assistant works internally, how to design reliable automation logic, how to integrate custom hardware and cloud platforms, and how to apply professional tools and best practices commonly used in IoT systems. The examples and projects presented in each chapter are based on solutions I have personally designed and implemented at home, making sure that they work. By the end of this book, you should be comfortable optimizing performance, developing custom device and integrations, securing and scaling up your setup, and combining local automation with cloud and AI-driven solutions.

This first chapter sets the foundation for everything that follows by providing a clear view of how Home Assistant works under the hood. Rather than focusing on installation or basic usage, it introduces the internal architecture, core components, and data flow that drives automations and system behavior. Understanding these fundamentals is essential for advanced topics explored later in the book. With this architectural context in place, you will be better prepared to make informed design decisions and get the most value from the chapters ahead.

You will begin by exploring the various components that constitute a Home Assistant installation, including the supervisor, core, apps, and integrations, and learning how they interact. The sequence will then progress to understanding data flow within Home Assistant, from sensor input to automation execution and state changes. A significant portion of the chapter will be dedicated to identifying and addressing performance bottlenecks. To wrap up

the chapter, I will provide a quick guide on how to find and fix performance issues in your Home Assistant installation.

We will also see a practical example of Home Assistant customization by creating an integration.

By the end of this chapter, you will be able to: understand Home Assistant's components, data flow and state machine, know how to create a Home Assistant integration, configure and manage Home Assistant's internal database effectively, and diagnose and resolve Home Assistant performance bottlenecks.

In this chapter, we will explore the following topics:

- Home Assistant behind the scenes: core components and interactions
- Understanding data flow and state management
- Performance profiling and optimization techniques

Note

Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Technical requirements

To complete the examples in this chapter, you will need:

- A computer running Windows, macOS, or Linux.
- Minimum recommended hardware: Intel Core i5 (or equivalent), 8 GB RAM (16 GB recommended), and at least 20 GB of free disk space.
- A Linux environment with Docker installed. If you are using Windows or macOS, you can run Linux in a virtual machine such as VirtualBox (<https://www.virtualbox.org/>).
- Visual Studio Code (VS Code) for editing files.
- A GitHub account to access the example code and create the custom integration.
- Basic familiarity with Docker, Git/GitHub, and the Linux command line.

- Basic knowledge of the vi editor, which is commonly used to edit files in Home Assistant OS.
- Basic knowledge of SQL will help you understand and customize the database queries presented in this chapter.

Some examples in this chapter assume familiarity with Docker, GitHub, Linux commands, SQL, and the vi editor. If you are not familiar with all of these tools, don't worry as the concepts required for the chapter are explained as they are introduced, and you can still follow the examples.

You can download the code for all examples in this book by following the instructions in the **Download the example code files** section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Home Assistant behind the scenes: core components and interactions

The Home Assistant architecture depends on the type of installation the user selects: **Home Assistant Operating System (HAOS)** or **Home Assistant Container**.

The HAOS installation has the essential components represented in *Figure 1.1*.

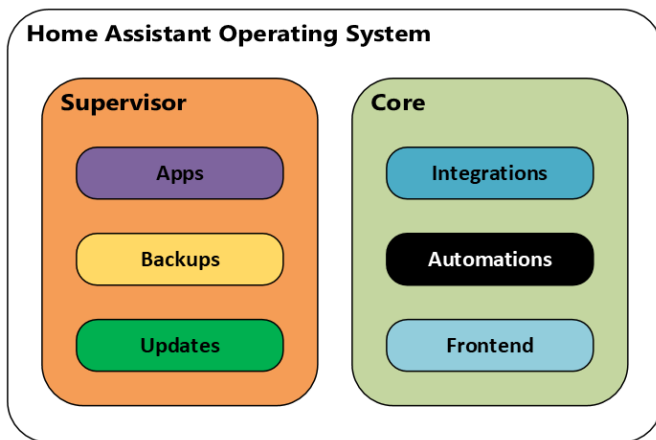


Figure 1.1 – Main components of Home Assistant Operating System installation architecture

The **Home Assistant Operating System** installation is a complete package that runs directly on dedicated hardware. As can be seen in *Figure 1.1*, It includes two main parts: **Supervisor** and **Core**. The **Supervisor** manages the system by controlling **Apps** such as MQTT broker, Node-RED, or InfluxDB, while also taking care of **Backups** and **Updates** to keep installations secure

and up to date. Alongside the **Supervisor**, the **Core** provides the main Home Assistant functionality. Within **Core**, users can configure **Integrations** to connect devices and services, create **Automations**, and access the **Frontend**, which delivers the user interface. This installation is designed as an all-in-one solution that simplifies management and reduces maintenance effort.

On the other hand, the Home Assistant Container Installation can be represented with the following diagram in *Figure 1.2*.

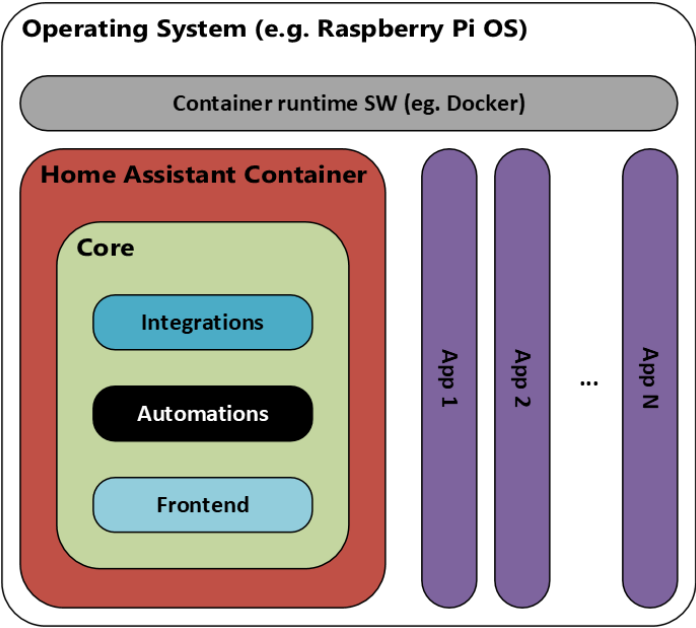


Figure 1.2 – Main components of Home Assistant Container Installation

Figure 1.2 shows that **Home Assistant Container** installation runs on a standard operating system such as Raspberry Pi OS, Debian, or Ubuntu, and requires container runtime software like Docker to host the application. Inside the **Home Assistant Container**, the **Core** is responsible for providing all of the essential functions, including **Integrations**, **Automations** and the **Frontend**. Unlike the full Operating System installation, this setup does not include the **Supervisor**, meaning there is no built-in support for **Backups**, **Updates** or official **Apps**. Instead, additional services must be installed and managed separately as independent **App** containers. These Apps run alongside the **Home Assistant Container** but remain the user's responsibility to configure and maintain. This installation type offers more flexibility but requires greater manual setup and management.

Table 1.1 summarizes the differences between the two installations:

Installation	Includes	Recommended Use	Performance
HAOS (Home Assistant Operating System)	Supervisor, Core, Apps, Backups	Raspberry Pi, Home Assistant Green, or mini PCs where Home Assistant is the primary application	Native performance.
Container (Docker)	Core only	Existing Linux servers running multiple Docker applications (for example, Plex, Frigate, InfluxDB, Nextcloud) where Home Assistant is one of several services	Performance is comparable to HAOS on the same hardware.

Table 1.1 – Home Assistant installations comparison

We will not go into the details of how to do each installation. Please refer to my first book *Building Smart Home Automation Solutions with Home Assistant* or *Home Assistant documentation* (<https://www.home-assistant.io/installation/>) if you wish to know more about both installations. What we are going to do instead is to take a deep dive into the main components of Home Assistant, namely HAOS, Supervisor, Apps, Core, and Integrations. Let's begin.

Home Assistant operating system (HAOS)

The **Home Assistant Operating System (HAOS)** is an optimized Linux environment customized to run **Home Assistant Supervisor** and **Core**. It is based on **Buildroot** (<http://buildroot.org>). **Buildroot** is a tool used to generate embedded Linux systems. It includes a cross-compilation toolchain, root filesystem generation, kernel image compilation, and bootloader compilation. Using Buildroot, Home Assistant can run on different hardware architectures, mainly based on Arm and x86 processors.

HAOS is a "locked-down" and purpose-built operating system. As opposed to a general-purpose Linux distribution like Raspberry Pi OS, you don't have full administrative control to install new drivers or compile the kernel from scratch. The system is designed to be stable, secure, and easily managed through the web interface.

One practical implementation to create your own custom Home Assistant OS image is to enable the native serial port on a Raspberry Pi 4. This can be useful in case you want to connect a device that communicates using the UART protocol. *Video 1.1* (<https://packt.link/SOF0U>) teaches you step-by-step how to do it.

Note

It is also possible to enable the Raspberry Pi I2C ports following the instructions here: <https://www.home-assistant.io/common-tasks/os/#enable-i2c>

Supervisor

Home Assistant Supervisor is a crucial component that manages the operating system, handles apps, updates, and provides a robust environment to run the Home Assistant Core. It is also responsible for making and restoring backups and creating a unified system that allows Home Assistant Core and apps to play audio.

One way to interact with Home Assistant Supervisor is to use the Supervisor API, which includes commands that can be issued by scripts. A list of endpoints and Models used by the Supervisor API is available here: <https://developers.home-assistant.io/docs/api/supervisor/endpoints> and here: <https://developers.home-assistant.io/docs/api/supervisor/models>.

Here is an example of a command issued from Home Assistant console to trigger a full backup in Home Assistant using the Supervisor API endpoint:

```
curl -s \
  -H "Authorization: Bearer ${SUPERVISOR_TOKEN}" \
  -H "Content-Type: application/json" \
  -X POST \
  -d '{"name": "My Full Backup"}' \
  http://supervisor/backups/new/full
```

Apps

Apps (formerly called add-ons) in Home Assistant let you expand its capabilities beyond the built-in features. They can run supporting applications that integrate with Home Assistant such as an MQTT broker or provide services like Samba file sharing so you can easily edit configuration files from other computers.

In *Chapter 11*, we will create our own App to manage a sensor we will create in the Custom integration development section in this chapter. You can also create your own app by following the tutorial available at Home Assistant official documentation: <https://developers.home-assistant.io/docs/apps/tutorial>.

Core

Besides Integrations, Automations and Frontend, the **Home Assistant Core** is built around four other primary components, supported by numerous helper classes that handle common tasks such as providing entities or managing location data. These four components are the following:

- **Event Bus:** The central communication channel that allows events to be fired and listened for.
- **State Machine:** Tracks the current states of all entities and triggers a `state_changed` event whenever a state update occurs.
- **Service Registry:** Monitors the event bus for `call_service` events and enables other parts of the system to register service actions.
- **Timer:** Emits a `time_changed` event every second on the event bus to drive time-based automations and updates.

A simplified diagram for the Core components mentioned before is represented in *Figure 1.3*.

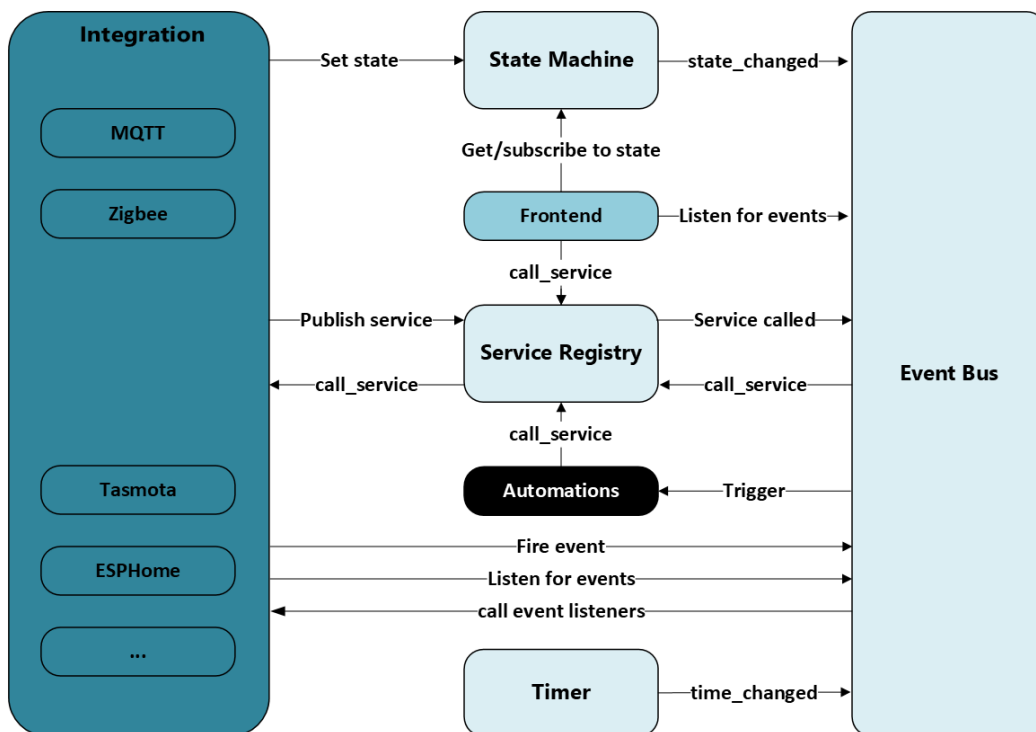


Figure 1.3 – Core components interconnection block diagram

For now, we will limit ourselves to the diagram for the Core. We will go into more detail on how some of the core components interact with other system components later in this chapter in the *Understanding data flow and state management* section.

Integrations

Integrations are part of the Home Assistant Core system. Home Assistant Core can be extended through integrations, each designed to handle a specific domain or type of device. An integration can listen for or trigger events, provide actions, and keep track of states. Technically, an integration is built from a component (the core logic) and one or more platforms (the parts that connect it to devices or services). Integrations are written in Python, which gives developers flexibility and access to a wide range of existing libraries and tools.

Types of integrations in Home Assistant

Integrations in Home Assistant cover a wide range of use cases, from connecting your lights and sensors to exposing new kinds of data or user actions. You can think of them in four main categories:

- **Internet of Things (IoT) device integrations:** These are integrations that connect to smart home hardware, such as lights, switches, sensors, or thermostats. They often use protocols like Zigbee, Z-Wave, or Wi-Fi.
- **External device and service integrations:** Integrations to communicate with cloud services or external APIs. For instance, the Spotify integration lets you control your playlists and speakers, and the Google Calendar integration allows automations to react to upcoming events in your schedule.
- **Virtual or computed entity integrations:** Integrations that create virtual sensors or helpers that provide derived or computed information.
- **Action and event-driven integrations:** These integrations trigger automations when an event occurs, such as a button press, a scheduled time, or a new email. For example, the IFTTT integration allows Home Assistant to react to events from external services.

Beyond these four types, there is also a difference in how integrations are configured or distributed: Custom Integration and Core Integration. *Table 1.2* summarizes the differences between these two types.

Topic	Custom Integration	Core Integration
Location	Installed locally under config/ custom_components/ your_integration/	Officially included in homeassistant/ components/your_integration/

Topic	Custom Integration	Core Integration
Distribution	Shared via folder, GitHub, or HACS	Bundled with every Home Assistant release
Development Speed	Rapid prototyping, no formal review	Must follow strict guidelines and review process
Flexibility	Full freedom to implement features	Must comply with async patterns, config flow, and coding standards
Testing and Stability	No continuous testing; may break after updates	Continuous integration testing ensures compatibility
Visibility	Discoverable only if shared; manual install often required	Immediately available to all users through the UI
Support and Maintenance	Maintained solely by the developer	Supported by the Home Assistant core team + community
Updates	Developer-controlled release schedule	Released in sync with Home Assistant updates
Documentation	Optional, usually in README or HACS listing	Required in official Home Assistant docs
Best Use Case	Niche devices, personal experiments, or fast prototyping	Broadly useful devices/services needing long-term stability

Table 1.2 – Main differences of custom and core integrations

Developing an integration

In *Chapter 3* of my first book, *Building Smart Home Automation Solutions with Home Assistant*, we built our own sensor, called the **Double Measurement Sensor**. There are two physical sensors in one enclosure, which outputs two different MQTT topics, one for temperature and another for motion. In this book, we will develop a custom Home Assistant integration for that sensor. Instead of configuring separate temperature and motion entities in `configuration.yaml`, the

integration allows you to configure a single device through the Home Assistant user interface. Throughout this exercise, you will learn how the main files of a Home Assistant integration work together and how to develop, configure, and test a custom integration before adapting it into a Core integration.

Before going into the details of integration development, let's prepare the development environment.

Follow the instructions provided on the Home Assistant website: https://developers.home-assistant.io/docs/development_environment. *Video 1.2* (<https://packt.link/Rd6BF>) explains how this is done, so we will not cover the detailed steps here.

We will start the Integration development with the custom integration and then adapt the code and indicate the steps to do for a core integration.

Custom integration development

We will start the development of our Custom Integration creating the basic skeleton files for it. Assuming you have the Core development environment setup in your VS-Code, run the command `python3 -m script.scaffold` in the terminal. The script integration will prompt you with some questions. Here's what we need to answer to develop the application in question:

- **Domain:** `dual_measurement_sensor`
- **Name:** Dual Measurement Sensor
- **Config flow:** `y`
- **IoT class:** `local_push`
- **Discovery:** `mqtt`
- **Dependencies:** `mqtt`
- **Requirements:** `(none)`
- **Codeowners:** `@your_github_id`
- **Platforms:** `sensor, binary_sensor`

It will create the following file structure:

- `custom_components/dual_measurement_sensor/`
- `__init__.py`
- `manifest.json`
- `config_flow.py`

- `const.py`
- `sensor.py`
- `binary_sensor.py`
- `strings.json`
- `translations/en.json`
- `README.md`

The `dual_measurement_sensor` integration follows the standard Home Assistant custom component structure, with each file serving a specific role in bringing together configuration, logic, and user interaction. At the center of the integration is the `__init__.py` file, which is responsible for initializing the component and registering the entities that will be created. This ensures that when Home Assistant starts, the integration automatically creates the temperature and motion sensor entities. For example, if the user has already configured a Dual Measurement Sensor through the Home Assistant interface, both entities are created and become available immediately after Home Assistant finishes starting. Here's the pseudocode for the `__init__.py` file created:

```
__init__.py

async_setup():
    return True
# required boilerplate for YAML-based setups, even if unused

async_setup_entry():
    hass.data[DOMAIN][entry_id] = {} # store entry-specific context
    forward entry setup to sensor.py and
    binary_sensor.py # delegate entity creation

async_unload_entry():
    cleanup hass.data # remove entry from memory
    unload platforms # remove entities from Home Assistant
```

Every integration must also include a `manifest.json` file. This is a metadata descriptor that tells Home Assistant the name of the integration, its version, which platforms it supports, and any dependencies or Python libraries it requires. Without this file, Home Assistant would not

be able to recognize or validate the integration. Here is the json formatted code for the `manifest.json` file.

```
{
  "domain": "dual_measurement_sensor",
  "name": "Dual Measurement Sensor",
  "version": "0.1.0",
  "config_flow": true,
  "dependencies": ["mqtt"],
  "iot_class": "local_push"
}
```

To allow users to configure the integration easily from the Home Assistant interface, a `config_flow.py` file is included. This file defines the interactive setup process, so instead of editing YAML, the user can enter information such as the device ID or MQTT topics directly through the graphical user interface. The configuration is then stored as an entry that Home Assistant can manage. Supporting this, a `const.py` file is used to store all constants in one place. Things like domain names, default values, and configuration keys are centralized here to ensure the code remains clean and consistent.

```
# config_flow.py
step_user():
    if user_input:
        create_entry with device_id + topics # save values in HA
    else:
        show config form # prompt user for required data

options_flow():
    allow changing debounce_ms, hold_ms, temp_window later # fine-tune after
    setup

# const.py
DOMAIN = "dual_measurement_sensor" # integration domain
CONF_TEMP_TOPIC = "temp_topic"
CONF_PRESENCE_TOPIC = "presence_topic"
DEFAULT_HOLD_MS = 2000 #default hold time for presence (ms)
```

The actual functionality of the integration is split across two files: `sensor.py` and `binary_sensor.py`. The `sensor.py` file creates the temperature entity, using Home Assistant's `SensorEntity` class as a base and updating its state when new MQTT message data is received. The `binary_sensor.py` file creates the presence detection entity, using the `BinarySensorEntity` class to represent ON/OFF states that indicate occupancy.

```
# sensor.py
on MQTT message:
    parse payload (float or JSON path) # extract temperature          value
    update moving average # smooth noisy readings
    if change > min_delta: # only update if significant
        update state in HA # publish new temperature value

# binary_sensor.py
on MQTT message:
    parse payload (on/off, true/false, 1/0) # flexible input parsing
    apply debounce + hold_ms # stabilize noisy presence signals
    update entity state ON/OFF # reflect occupancy in HA
```

To make the integration accessible and user-friendly, Home Assistant requires localized strings that can be displayed in its interface. The `strings.json` file defines the base set of messages, titles, and labels used in the integration's configuration flow. Translations are then provided in separate language files such as `translations/en.json`, which maps those keys to English text. Additional languages can be added in the same way, making the integration fully translatable for international users.

```
{
  "title": "Dual Measurement Sensor",
  "config": {
    "step": {
      "user": {
        "title": "Add Dual Measurement Sensor",
        "description": "Configure MQTT topics and options for your device.",
        "data": {
          "device_id": "Device ID",
          "temperature_topic": "Temperature topic",
          "presence_topic": "Presence topic",
          "temperature_json_path": "Temperature JSON path",
          "temperature_window": "Temperature window (samples)",
          "temperature_min_delta": "Minimum delta to publish (°C)",

```

```
"presence_debounce_ms": "Presence debounce (ms)",
    "presence_hold_ms": "Presence hold (ms)" }
    }
  },
  "abort": {
    "already_configured": "This device is already configured."
  }
}
```

Finally, the repository includes a `README.md` file, which serves as the documentation for the integration.

Note

A brand for your Integration can be created using a logo and icon image file with specific format and sizes. They need to be committed and Pull requested to <https://github.com/home-assistant/brands>.

Next we will move all these files created to `config/homeassistant/custom_components/dual_measurement_sensor` and reboot the system. To configure devices to use the integration simply go to **Settings | Devices & Services | Add Integration**. Search for **Dual Measurement Sensor** and click on it. The configuration screen shown on the left of *Figure 1.4* will be presented. Fill the information properly with the data from your sensor and click **Submit**. You can check later how the device using the integration will look by looking for **dev1** in the **Devices** tab. It will look like the integration shown on the right of *Figure 1.4*.

The screenshot displays two panels from the Home Assistant interface. The left panel, titled 'Add Dual Measurement Sensor', is a configuration form. It includes fields for 'Device ID*' (dev1), 'Temperature topic*' (tele/Garage_Temp_PIR/SENSOR), 'Presence topic*' (sensor/Garage_Temp_PIR/PIR1), and 'Temperature JSON path (e.g. BMP280.Temperature)' (BMP280.Temperature). Below these is a 'Temperature window (samples)' slider set to 1, a 'Minimum delta to publish (°C)' field set to 0.1, a 'Presence debounce (ms)' field set to 300, and a 'Presence hold (ms)' field set to 5000. A 'SUBMIT' button is at the bottom. The right panel, titled 'dev1', shows the device's presentation. It includes a 'Device info' card with the name 'Dual Measurement Device' and manufacturer 'marcoascarvalho'. A 'Sensors' card shows 'Presence' and 'Temperature' (34.4 °C) with an 'ADD TO DASHBOARD' button. 'Automations', 'Scenes', and 'Scripts' cards are also present, each with a '+' button and a message stating no items have been added. A 'Logbook' card on the right shows a list of events for August 23, 2025, including 'dev1 Presence cleared (no motion detected)' and 'dev1 Presence detected motion' at 6:16:59 PM, 6:16:58 PM, 6:16:52 PM, and 6:16:19 PM.

Figure 1.4 – Dual Measurement Sensor Integration configuration and presentation

All code developed for the dual measurement sensor integration is available in the downloadable code bundle under the Chapter 1 folder. You can download the code by following the instructions in the Download the example code files section in the Preface.

Turning a custom integration to a core integration

As mentioned before, a core integration must follow a meticulous and strict protocol to be integrated into the official release of Home Assistant. Here is what you have to do to turn your custom integration into a core integration:

- Remove version from `manifest.json`
- Ensure that the documentation link in `manifest.json` points to: `https://www.homeassistant.io/integrations/your_integration_name`
- Verify that the files reside in `homeassistant/components/your_integration_name`

- Instead of a `strings.json`, Core integration requires a `translations/en.json`. Content can be identical to your `strings.json`, just rename and put it in a `translations/` folder.
- Icons and logos should be included as commented before but you have to redo the process to include the files in the `core_integrations/dual_measurement_sensor` instead of `custom_integrations/dual_measurement_sensor`.
- Tests should be developed and uploaded to `homeassistant/tests/components/your_integration_name`

After you do all the steps above, it is required to review the Integration Quality Scale and fill in the checklist so your integration can be rated at least in the Bronze category. More information about the quality scale can be found here: https://www.home-assistant.io/docs/quality_scale/

Frontend

The Home Assistant frontend is the user-facing layer that turns entities, automations, and diagnostics into an accessible, responsive UI—dashboards, dialogs, editors, and visualizations that the household end user touches.

As a developer, you can contribute with new cards, panel views, and configuration editors; improve performance, accessibility, and localization; or extend integration UIs with options flows and richer diagnostics. The stack is modern web, including TypeScript, Lit and Web Components. It talks to Core over the WebSocket/HTTP APIs, so you can prototype quickly and release features that feel native.

You can start the Frontend development with the official docs at <https://developers.home-assistant.io/docs/frontend> and align your UX with the Home Assistant design system (components, spacing, typography, states) at <https://design.home-assistant.io> to ensure your contributions look and behave consistently across the platform.

Understanding data flow and state management

We will better understand the data flow and state management in Home Assistant using *Figure 1.5*. The sequence represented is the classical example of what happens behind the scenes of a motion-activated light.

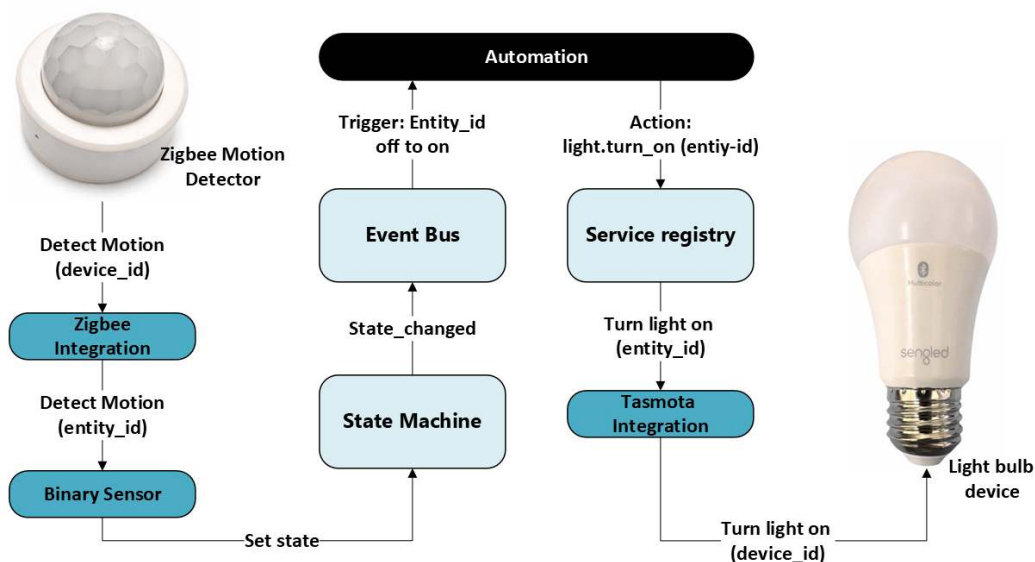


Figure 1.5 – Sequence of data flow of a motion-activated light automation

Here is a detailed description of the blocks and pictures in *Figure 1.5*:

The Physical Devices are the following:

- **Zigbee Motion Detector:** This is the physical sensor that detects motion. It sends a Zigbee message when motion is detected. The label `device_id` indicates that it belongs to a single hardware device.
- **Light bulb device:** This is the physical light bulb that is controlled by the system.

The Home Assistant integrations, shown as blue boxes, are the following:

- **Zigbee Integration:** This integration is responsible for communicating with the Zigbee Motion Detector. It receives the Zigbee signal indicating motion. Its role is to convert this hardware signal into a format that Home Assistant can understand.
- **Binary Sensor:** This is a conceptual entity type within Home Assistant. The Zigbee Integration doesn't just pass the raw data; it uses Home Assistant's built-in Binary Sensor component to create a software entity. This entity will have a state that is either on or off (in this case, on for "motion detected").
- **Automation:** This represents the automation rule that the user has created. It's an internal Home Assistant component that manages actions based on triggers. The diagram shows its key elements:

- **Trigger:** This specifies that the automation will run when the Binary Sensor entity's state changes from off to on.
- **Action:** Calls service `light.turn_on(entity_id)`. This is the action that the automation will take when triggered. It makes a service call to turn on a specific light, identified by its `entity_id`.
- **Tasmota Integration:** This integration is responsible for communicating with the Light bulb device. It registers a `light.turn_on` service with the Home Assistant Core so that other components can control the light.

The Home Assistant Core components or light blue Boxes are the following:

- **State Machine:** This is the single source of truth for the state of all entities. The diagram shows the flow:
- **Set state:** The Binary Sensor entity's state is updated (Set state) in the State Machine.
- **State_changed:** When the state changes, the State Machine fires a `State_changed` event.
- **Event Bus:** This acts as a communication backbone. When an event is fired like `State_changed`, it is broadcast on the Event Bus. Any component listening for that specific event can react. The Automation component is a key listener.
- **Service Registry:** This is a look-up table for all available services. When an Integration like Tasmota wants to provide the turn-on service, it registers the service request here. When another component, such as an Automation, wants to call a service, it goes through the Service Registry as well.

The sequence of the end-to-end flow from the left to right in the diagram presented in *Figure 1.5* can be explained as the following steps:

1. **Motion Detected:** The physical Zigbee Motion Detector senses motion.
2. **State Updated:** The Zigbee Integration receives the signal and updates the Binary Sensor entity's state from off to on in the State Machine.
3. **Event Fired:** The State Machine recognizes the state change and fires a `State_changed` event onto the Event Bus.
4. **Automation Triggered:** The Automation is listening on the Event Bus. It sees the `State_changed` event and confirms that the conditions of its trigger (`entity_id` off to on) are met.
5. **Service Called:** The automation's action is executed. It makes a Call service request for `light.turn_on`, identifying the specific `entity_id` of the light, and sends this request to the Service Registry.

6. **Integration Invoked:** The Service Registry looks up the `light.turn_on` service and routes the request to the correct handler, which is the Tasmota Integration.
7. **Physical Command Sent:** The Tasmota Integration receives the command. It translates the Home Assistant `entity_id` into a specific command (Turn on) and device identifier (`device_id`) that the Tasmota firmware understands. It then sends this command to the Light bulb device.
8. **Light On:** The physical light turns on.

With a clear understanding of how data flows through Home Assistant you can begin to see how even simple automations rely on many internal components working together. As the number of devices, integrations, and automations grows, this internal activity increases, placing additional load on the system. At this point, understanding what happens inside Home Assistant is no longer enough — it becomes important to understand how efficiently it happens. In the next section, we shift focus from architecture to practice by exploring performance profiling and optimization techniques that help keep Home Assistant responsive, stable, and scalable as your system evolves.

Performance profiling and optimization techniques

Home Assistant is highly flexible, but with flexibility comes the challenge of maintaining good performance as your system grows. A system with dozens of integrations, thousands of entities, and years of historical data can quickly slow down if left unchecked. In this section, we will explore techniques to diagnose bottlenecks and apply optimizations that ensure your Home Assistant remains fast, stable, and scalable.

Database optimization

Database optimization could be required if you have multiple devices in your home logging information to the Home Assistant database. As you add more devices, you may not notice how frequently they are recording data or whether all of that information needs to be stored. In general, you should consider optimizing the database if you notice slower startup times, delayed automations, slower History or Logbook views, or if the database grows to several gigabytes in size. It's important to review the recorded data so you can avoid:

- Growing your Home Assistant database, which can reduce space allocated for the system.
- With less space in the system, I/O operations become slower, degrading system performance.

- Depending on the frequency of the data being inserted in the database, the system will be busy multiple times, degrading system performance as well.
- With many I/O operations being executed, some memory devices like SD-cards can fail prematurely due to excessive I/O operations performed.

In other words, database optimization could be translated into improved system performance, reduced memory usage, decreased I/O usage, and extended memory device lifetime.

Here are the steps required for database optimization:

1. Identify the entities that contribute to the most entries in the database.
2. Verify if the information logged in the system is important for you and if you need the current data rate to be logged.
3. Configure, filter, or exclude device data.

Home Assistant stores all information related to states and events in the `home-assistant_v2.db` file located in the `config/homeassistant` directory. You can inspect and query the database using the SQLite Web app, as shown in *Figure 1.6*. Before making any changes to the database, create a backup of the `home-assistant_v2.db` file. If a query or modification accidentally corrupts or deletes data, the backup allows you to restore the database and avoid losing historical information.

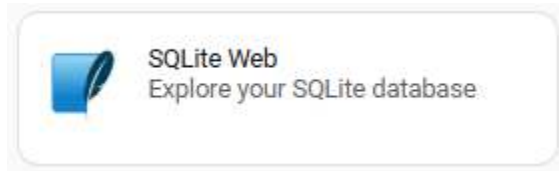


Figure 1.6 – SQLite Web app icon

Note

To install SQLITE Web, you need to enable the **Advanced** mode in the **User Settings** area.

One of the first things you can do to evaluate how large your database is, is check the size of your states and events tables using the following commands:

```
SELECT COUNT(*) FROM states;  
SELECT COUNT(*) FROM events;
```

In my case, these values were 407,797 and 8,188, respectively, at the time of writing this chapter. Next, we will proceed with the steps to optimize the database.

Identifying noisy entities and events

Some entities like temperature sensors updating every few seconds or rapidly toggling motion sensors can flood the database with unnecessary records. Excluding or reducing their data input in the database improves responsiveness and reduces disk I/O. You can use the following SQL query in the SQLite Web to look for the 10 entities that are creating more events in a day:

```
SELECT sm.entity_id, COUNT(*) AS rows_1d
FROM states s
JOIN states_meta sm ON sm.metadata_id = s.metadata_id
WHERE s.last_updated_ts >= strftime('%s','now','-1 days') GROUP BY sm.entity_id
ORDER BY rows_1d DESC
LIMIT 10;
```

Note

Depending on which Home Assistant version you are using, your recorder database schema may differ. If you see `entity_id` directly in the `states` table, you can query it directly. If you only see `metadata_id`, then join with `states_meta` to retrieve the entity name.

The query gave me the results shown in *Figure 1.7*:

Results (10) [Permalink](#)

entity_id	rows_1d
sensor.z_computer_outlet_power	9343
sensor.desk_outlet_current_consumption	7226
sensor.desk_outlet_current	5553
sensor.desk_outlet_voltage	4521
sensor.arris_sbg7400ac2_router_kib_s_received	2861
sensor.arris_sbg7400ac2_router_kib_s_sent	2861
sensor.worldclock_sensor	1433
sensor.date_time	1432
sensor.total_energy_consumption_wifi	698
sensor.desk_outlet_today_s_consumption	411

Figure 1.7 – 10 entities that create more data entries in database

This means that the top sensor, `sensor.z_computer_outlet_power`, added 9343 entries in the database in one day, which is one entry per 9.25 seconds. After looking at the sensor configuration, I realized that this sensor outputs data every 5 seconds!

States are not the only entities that can inflate the database, events can also contribute to that. The following SQL query can retrieve the 10 events that contribute the most entries to populate your Home Assistant database:

```
SELECT et.event_type, COUNT(e.event_type_id) AS events_30d
FROM events e
JOIN event_types et ON e.event_type_id = et.event_type_id
WHERE e.time_fired_ts >= strftime('%s', 'now', '-30 day')
GROUP BY et.event_type
ORDER BY events_30d
DESC LIMIT 10;
```

Running the query just mentioned in my Home Assistant database gave me the result presented in *Figure 1.8*:

Results (10) [Permalink](#)

event_type	events_1d
recorder_5min_statistics_generated	3095
service_registered	621
call_service	312
component_loaded	288
recorder_hourly_statistics_generated	258
automation_triggered	233
entity_registry_updated	75
device_registry_updated	52
panels_updated	33
script_started	11

Figure 1.8 – 10 events that create more data entries in database

The major event `recorder_5min_statistics_generated` added 3093 entries in 30 days, which is less than 5 events per hour. Compared to states, events generated much less data in the database in the case of my configuration.

Database maintenance: filtering undesired information

In the query output example presented in *Figure 1.7*, let's filter, configure, or exclude the data from the top 6 devices so we can have less information being written in the database, and avoid the consequences mentioned earlier.

The two top devices presented in *Figure 1.7* are smart plugs with energy monitoring. The `sensor.z_computer_outlet` is a zigbee device from ThirdReality and the `sensor.desk_outlet` is a TP115 Wifi device from Kasa. Unfortunately, at the time of writing this book, there is no official support available to configure the reporting data rate for these plugs. So, in this case, it will not be possible to configure these devices to report data at a different rate. What are we going to do instead? The cleanest fix in Home Assistant is to reduce what gets recorded. The recorder integration filters unwanted data from being stored in the database. It also allows you to control how long historical data is retained, following the same data retention principle used in many database systems to balance storage usage and performance.

We can use some pragmatic approaches using the recorder integration to optimize the database performance and space:

1. Purging data from the database after a certain period, enabling controlled growth of it.
2. Excluding high-rate report metrics entirely from the recorder. This means keeping the entity live in the User Interface but not storing every state change or event.
3. Publishing a "filtered" companion sensor that only updates on meaningful changes like greater than 10 W delta, or once every 5 minutes as a keep-alive, then record only the filtered sensor.

The first approach is done by the following YAML code added to the `Configuration.yaml` file:

```
recorder:
  purge_keep_days: 30
  auto_purge: true
  auto_repack: true
```

It is implemented by adding the code after the word `recorder`. Change the `purge_keep_days` for the number of days you need. In the above code just shown, any state or event older than 30 days is deleted during the automatic cleanup job. A lower number means smaller database retention, which translates into a smaller database size, faster queries, and better performance.

The `auto_purge` enables automatic purge. Otherwise, you'd have to call the `recorder.purge` service manually. With it set to `true`, Home Assistant will regularly clean up rows older than `purge_keep_days`.

After purging, the database file may still take up the same disk space as old pages aren't reused. The `auto_repack: true` tells Home Assistant to run a cleanup database operation, reclaiming free space and making the DB smaller and faster. This prevents it from growing over time.

The second approach is executed by the following code added to the `configuration.yaml` file:

```
Recorder:
  exclude:
    entities:
      - sensor.desk_outlet_current_consumption
      - sensor.z_computer_outlet_power
      - sensor.desk_outlet_voltage
      - sensor.desk_outlet_current
      - sensor.arris_sbg7400ac2_router_download_speed
      - sensor.arris_sbg7400ac2_router_upload_speed
      - sensor.date_time
      - sensor.worldclock_sensor
```

The second approach to be implemented is to exclude some unnecessary entities from being recorded. From our list in *Figure 1.7*, let's remove the entities related to world clock, datetime, router 7400 and others related to `sensor.z_computer_outlet` and `sensor.desk_outlet` not need after implement the third approach explained next. The portion of the code that remove the entities to be recorded in the database is mentioned after the word `exclude` in the code just presented.

The third approach is implemented by adding the following code to `Configuration.yaml` as follows:

```
template:
  - trigger:
      - platform: state
        entity_id:
          - sensor.desk_outlet_current_consumption
          - sensor.z_computer_outlet_power
      - platform: time_pattern
        minutes: "/5"
    sensor:
      - name: "Desk Outlet Power (filtered)"
        unique_id: desk_outlet_power_filtered
        unit_of_measurement: "W"
        state: >
```



```

    {% set raw = states('sensor.desk_outlet_current_consumption')|float(0)
%}

    {% set prev = this.state|float(0) %}
    {% set changed = (raw - prev)|abs >= 10 %}
    {{ raw if changed or trigger.platform == 'time_pattern' else prev }}
- name: "Computer Outlet Power (filtered)"
  unique_id: computer_outlet_power_filtered
  unit_of_measurement: "W"
  state: >
    {% set raw = states('sensor.z_computer_outlet_power')|float(0) %}
    {% set prev = this.state|float(0) %}
    {% set changed = (raw - prev)|abs >= 10 %}
    {{ raw if changed or trigger.platform == 'time_pattern' else prev }}

```

The third approach is to create filtered sensors that only update when there is a change of 10 W or more, or at least every 5 minutes as a keep-alive, for the entities that have a high data rate in the database. In our example, these entities are `sensor.z_computer_outlet` and `sensor.desk_outlet`. The code that implements this approach is after the word `template`. I'll leave the explanation of the template code for *Chapter 3* when we will discuss the use of Templates.

Implementing these techniques will ensure better database performance in terms of speed and size, as discussed above.

Logging configuration

Log data doesn't affect the recorder database, but lower log volume translates to fewer disk writes, which is nice for SD cards like the ones I used on my Raspberry Pis. In Home Assistant, the logger integration controls how much detail goes into your log.

The following YAML code can be included in the `Configuration.yaml` file to reduce the quantity of information on logs:

```

logger:
  default: warning
logs:
  homeassistant.components.mqtt: info
  custom_components.dual_measurement_sensor: debug

```

The `default: warning` option makes sure all components will have the warning level for logs. The `logs` option lets you override this per component — for example, configuring MQTT to log

state changes and service calls. For the Dual Measurement Sensor integration we developed earlier in the *Developing an integration* section, we can set it to log every detail in debug mode.

Another approach we can take with regard to logging is to redirect logs to external storage like SSDs, to reduce wear on local disks.

Integration minimalism

Every active integration consumes CPU, RAM, or disk resources. Only enable integrations that you actually use. Disable or remove integrations for devices or services that are no longer needed.

Note

Periodically review **Settings | Devices & Services** and remove unused integrations.

Resource monitoring

Another way to discover performance bottlenecks is resource monitoring, that is, checking the status of CPU, memory, or I/O.

You can do this by installing the **Glances** tool available in the community app store:

1. Go to **Settings | Apps**. Click on the **Install app** button.
2. Search for glances. Click on the icon presented in *Figure 1.9*.
3. Install the glances app by clicking on the **Install** button.

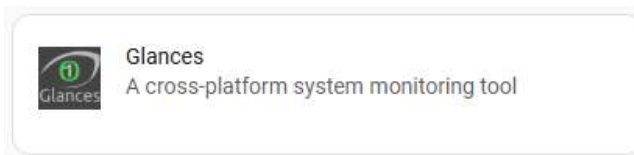


Figure 1.9 – Glances app icon

When you run Glances, you get a complete system monitor dashboard, displaying CPU, memory, load average, swap, disk usage, disk I/O, network I/O, and top processes — all in one convenient overview. A Glances screenshot is presented in *Figure 1.10*:

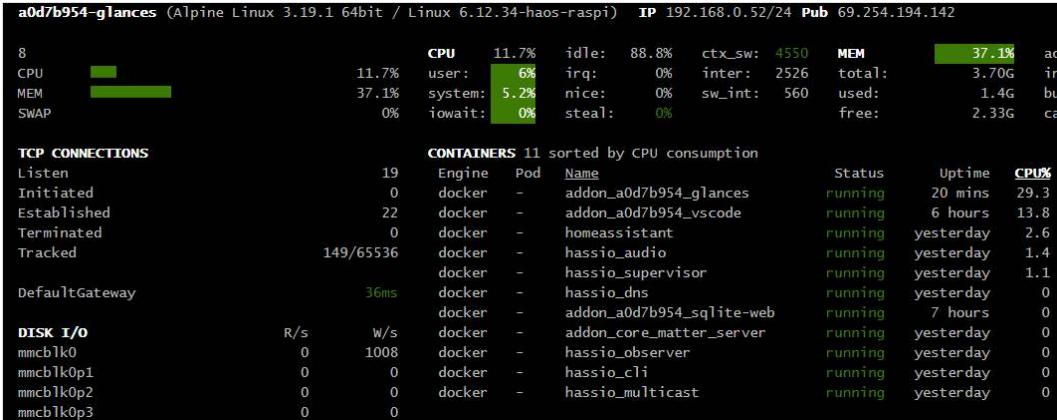


Figure 1.10 – Glances screenshot – monitoring resource tool

Note

You must disable the protection mode of the **Glances** application to be able to use it.

Glances highlights in green the performance or utilization of certain resources if it is acceptable; otherwise, the color changes to red. You can easily verify how your system is performing using this color code.

Resource monitoring offers visibility into how Home Assistant is using CPU, memory, and disk I/O, but these metrics alone do not always explain why a system feels slow. Performance issues often require correlating resource usage with system behavior, integrations, and recent changes. In the following practical example, we apply a structured diagnostic approach that combines monitoring data with targeted analysis to identify and resolve a slow Home Assistant system.

Practical example: diagnosing a slow system

Imagine that your Home Assistant automations begin to trigger with a noticeable delay. Instead of immediately reinstalling or replacing hardware, you can follow the diagnostic process represented in *Figure 1.11* to identify and resolve the underlying cause.

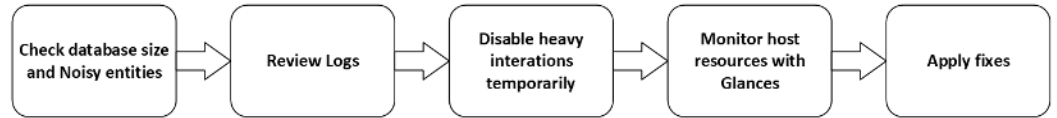


Figure 1.11 – Diagnostic process to discover and fix performance issues

By combining database inspection, log review, resource monitoring, and selective integration testing, you can create a consistent workflow for diagnosing and solving performance bottlenecks in Home Assistant.

Summary

In this chapter, we explored the architecture and internal mechanisms that are part of Home Assistant. Starting with the differences between installation types, such as Home Assistant OS and Container, we explored the core components like Supervisor, Core, Apps, and Integrations. We examined each component in terms of its role and interaction within the broader ecosystem. We also reviewed the data flow and state management within Home Assistant, using real-world automation examples to illustrate the path from sensor input to automation triggers and system reactions.

We looked at practical examples, such as customizing the Home Assistant OS and building a custom integration, to get hands-on experience with deeper system-level interactions. We explored performance profiling and optimization techniques to maintain system responsiveness and reliability.

By the end of this chapter, you should have a strong foundation in how Home Assistant works behind the scenes, how to build integrations, and how to identify and resolve performance issues in your setup.

In *Chapter 2, Mastering YAML for Complex Home Assistant Configurations*, we look at how to use advanced YAML features to organize configurations, manage complexity, and build reliable automations in Home Assistant.

2

Mastering YAML for Complex Home Assistant Configurations

YAML is the backbone of configuration in Home Assistant. It is the language that defines how devices are connected, how automations behave, and how your Home Assistant system configuration operates behind the scenes. The user interface allows you to configure many features, but YAML remains essential when you need full control and flexibility. As your system grows, poorly structured YAML configuration quickly becomes difficult to manage and troubleshoot, making it critical to understand not just the syntax, but how to use it effectively.

In *Chapter 1*, you explored how Home Assistant works internally, including its architecture, data flow, and core components. Now, you will build that foundation by focusing on how to control and shape that system through well-structured configurations. Mastering YAML at this stage is the logical next step, as it allows you to translate architectural understanding into practical, maintainable implementations.

In this chapter, you will understand YAML by learning how to write advanced and modular configurations, organize complex setups for readability and scalability, safely manage changes using version control, protect sensitive data, and troubleshoot common errors. These skills will enable you to confidently design and maintain robust Home Assistant environments.

By the end of this chapter, you should be able to write advanced and modular YAML configurations for Home Assistant, structure configurations for maintainability and scalability, implement version control with GitHub, manage sensitive data effectively using secrets .yaml, and troubleshoot and debug YAML configuration issues.

This chapter will cover the following topics:

- Advanced YAML syntax: anchors, aliases, and includes

- Modularizing your configuration: strategies for scalability
- Version control with Git: managing your Home Assistant configuration
- Secrets management and sensitive data handling
- Troubleshooting YAML: common pitfalls and debugging techniques

Technical requirements

Before starting this chapter, make sure you have the following:

- Familiarity with VS Code or the Home Assistant File Editor for editing YAML and other code files.
- Access to the SSH terminal in Home Assistant, since some examples require running command-line instructions.
- A GitHub account configured and ready to create repositories.
- Basic familiarity with GitHub workflows, such as creating repositories and managing configuration files.
- Basic knowledge of shell scripting, which will help you understand some of the scripts presented in this chapter.

You can download the code for all examples in this book by following the instructions in the **Download the example code files** section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Advanced YAML syntax: anchors, aliases, and includes

YAML, short for "YAML Ain't Markup Language," is a human-readable format designed for writing configuration files. Home Assistant uses YAML as its primary configuration language because its simple, indentation-based syntax makes configurations easy to read, write, and maintain. At the same time, YAML supports hierarchical data structures, such as lists, dictionaries, and nested mappings, making it well suited for defining entities, automations, scripts, and integrations in Home Assistant.

Another reason for its adoption is that YAML is already widely used in DevOps and infrastructure-as-code tools such as Ansible, Kubernetes, and Docker Compose. This means many developers and engineers are familiar with it. Finally, Home Assistant can parse YAML into its internal state machine efficiently, while still allowing overrides in the UI.

Some important YAML files in Home Assistant include:

- `configuration.yaml` – the main entry point for almost all setup.
- `automations.yaml` – where automations are stored if you prefer YAML over the UI editor.
- `scripts.yaml` – for custom scripts triggered by automations.
- `scenes.yaml` – to define lighting, media, or device group states.
- `secrets.yaml` – to store API keys and passwords securely.
- `custom YAML files` - user-defined files for any additional configuration.

Understanding advanced YAML syntax helps you reduce duplication, keep files modular, and make your configuration scalable as your system grows.

We will not go over all YAML syntax and style rules. You can check Home Assistant documentation to know more about that: <https://www.home-assistant.io/docs/configuration/yaml/>. What we will do instead is explore some of the useful and advanced YAML resources that will help you do more and better with your system.

Anchors and aliases

We will first explore the utilization of Anchors and Alias. Both are defined as follows:

- **Anchor** – lets you define a YAML block once and give it a name. It is represented in the YAML code as the letter `&`.
- **Alias** – references the anchor elsewhere in the file, reusing the same block. It is represented in the YAML code as the symbol `*`.

The idea behind the use of Anchors and Alias is to avoid repeating the same configuration multiple times. It is useful when you have entities that share a common configuration, and you want to reduce the replication of this configuration in each entity. You can define a common Anchor that includes all these shared settings, and then reference it with an Alias at each device.

In my book, *Building Smart Automations with Home Assistant*, we created a Five-zone Temperature Logger. We had to create a configuration in Home Assistant to group data from Bluetooth thermometers. As I recently discovered the use of Anchor and Alias, I decided to apply the concept to this thermometer configuration. The code snippet below shows what the new configuration looks like.

```
mqtt:
  sensor:
    - &bt_temperature_sensor
```

```

state_topic: "tele/ESP32_TEMPERATURE_HUB/SENSOR"
icon: mdi:thermometer-bluetooth
unit_of_measurement: "°C"
device_class: temperature

- <<: *bt_temperature_sensor
  name: "Backyard Temperature"
  value_template: '{{ value_json.ATC19f133.Temperature }}'

#
...

- <<: *bt_temperature_sensor
  name: "Master Bedroom Temperature"
  value_template: '{{ value_json.ATC5d22d1.Temperature }}'

```

In the code snippet presented, the Anchor `&bt_temperature_sensor` YAML code block is defined just one time, and the entire block is re-used using the Alias `<<:`

`*bt_temperature_sensor` five times to create the five temperature sensors.

Using this new configuration presented, I was able to reduce the code from 34 lines to 24 lines or around 30%. It will be more or less efficient, depending on the number of entities you have and the number of parameters to be configured in the Anchor. The higher the number of entities and Anchor parameters, the higher the number of code lines saved.

Includes

As your Home Assistant installation grows, keeping every entity, automation, script, and integration inside a single configuration `.yaml` file quickly becomes difficult. A large configuration file is harder to read, debug, maintain, and share with others. It also increases the chances of merge conflicts when using Git to track changes.

To address these challenges, Home Assistant provides several include directives that allow you to split your configuration into smaller, logical files. When Home Assistant starts, it reads all referenced files, merges them into a single configuration, validates the resulting structure, and then loads it into memory.

The benefits of using include directives are:

- Improved readability by organizing related configurations into separate files.
- Easier maintenance since each file has a specific purpose.
- Better scalability for medium and large Home Assistant installations.
- Simplified troubleshooting because configuration errors are isolated to smaller files.

Figure 2.1 illustrates how each include directive organizes one or more YAML files before Home Assistant merges them into the final configuration.

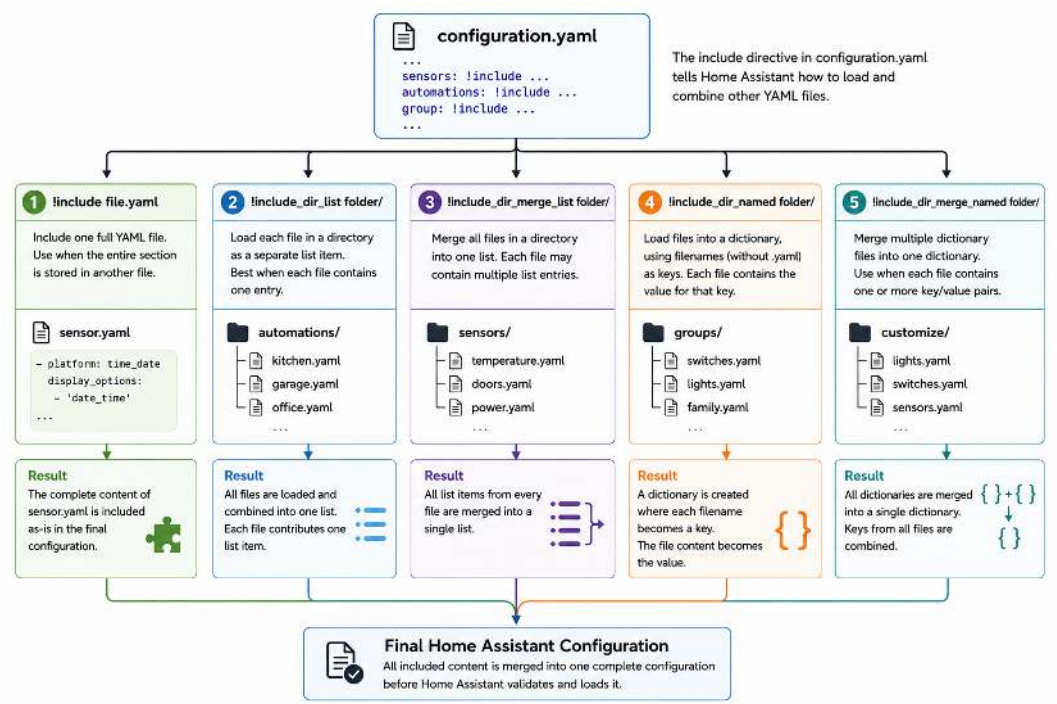


Figure 2.1 – Home Assistant include directives and how they merge configuration files

Table 2.1 summarizes the include directives supported by Home Assistant and their recommended use cases.

Directive	Best used when	Result
<code>!include file.yaml</code>	One configuration is stored in another file	Includes one complete YAML file
<code>!include_dir_list folder/</code>	Each file contains one list item	Creates one list containing every file
<code>!include_dir_merge_list folder/</code>	Each file contains multiple list items	Merges all list items into one list
<code>!include_dir_named folder/</code>	Filenames should become dictionary keys	Creates a dictionary using filenames as keys

Directive	Best used when	Result
<code>!include_dir_merge_named</code> folder/	Multiple dictionary files should become one dictionary	Merges all dictionaries into one

Table 2.1 – Include directives supported by Home Assistant

A dictionary is a collection of key-value pairs, where each key is a unique name used to identify a specific set of information. The key acts as a label, while the value contains the associated configuration or data. For example:

```
lights:
  entities:
    - light.kitchen
    - light.living_room

switches:
  entities:
    - switch.garage
    - switch.garden
```

In this example, `lights` and `switches` are dictionary keys. Everything indented beneath a key represents its associated value.

The following sections explain when to use each directive and demonstrate practical examples.

Directive `!include file.yaml`

Use this directive when an entire configuration section can be moved into its own file. It is the simplest include method and is ideal for separating large sections from `configuration.yaml`.

Home Assistant already uses this approach by default for automations, scripts, and scenes:

```
automation: !include automations.yaml
script: !include scripts.yaml
scene: !include scenes.yaml
```

Whenever you create a new automation, script, or scene from the Home Assistant user interface, it is stored in these files.

You can use the same approach for other integrations. For example, adding `sensor: !include sensor.yaml` to `configuration.yaml` moves the complete sensor configuration into `sensor.yaml`. *Table 2.2* shows the resulting structure.

configuration.yaml	sensor.yaml
<pre>sensor: - platform: time_date display_options: - 'date_time'</pre>	<pre>- platform: time_date display_options: - 'date_time'</pre>

Table 2.2 – Directive `!include file.yaml` example

Directive `!include_dir_list folder/`

Use this directive when each file contains a single configuration entry, and you want Home Assistant to combine all of them into one list. A common example is organizing automations by room. Instead of storing every automation inside one large file, your directory might look like this:

```
automations/
|
├─ kitchen_lights.yaml
├─ living_room.yaml
├─ garage.yaml
└─ backyard.yaml
```

Each file contains a single automation (one list item beginning with `-`). When Home Assistant starts, it automatically loads every file and creates one complete automation list.

You can further organize your automations by creating subfolders based on rooms, floors, or functions. For example, the kitchen folder could contain automations related to appliances, while the garage folder could contain automations for the garage door and lighting. The filenames are entirely up to you and should be descriptive enough to make the configuration easier to navigate and maintain.

```
automations/
|
├─ kitchen/
|   └─ coffee_machine.yaml
```

```
|   └─ toaster.yaml
|
| └─ garage/
|   └─ garage_door.yaml
```

Note

If you create automations using the Home Assistant Automation Editor, it is recommended to keep the default `automations.yaml` file because the editor expects a default location for newly created automations. The file can remain empty if you organize automations into separate files afterward.

Directive `!include_dir_merge_list folder/`

Use this directive when each file contains multiple list entries instead of just one. For example, you may want to organize sensors by category:

```
sensors/
|
| └─ temperature.yaml
| └─ doors.yaml
| └─ energy.yaml
```

Each file may contain multiple sensor definitions. Home Assistant reads every file and merges all entries into one large sensor list. Each item inside every file must begin with a hyphen (-) because they are list entries.

Directive `!include_dir_named folder/`

Use this directive when you want to organize configurations into separate files while allowing Home Assistant to automatically use each filename as a dictionary key. For example:

```
groups/
|
| └─ switches.yaml
| └─ lights.yaml
| └─ family.yaml
```

If `switches.yaml` contains

```
name: Living Room Lamps
```

```
entities:
  - switch.livingroomlamp1
  - switch.livingroomlamp2
```

It will become:

```
group:

  switches:

    name: Living Room Lamps

    entities:

      ...
```

Home Assistant creates the dictionary key using the filename (switches), making it easier to organize related configurations.

This approach works well when each file represents a distinct configuration category or object. For example, you can create separate files for lights, switches, media players, or family groups. Home Assistant uses the filename as the dictionary key, making the configuration easy to organize and maintain.

Directive `!include_dir_merge_named folder/`

Use this directive when you want to split a large dictionary-based configuration into multiple files while preserving a single dictionary in the final configuration. !

`include_dir_merge_named` merges the dictionaries contained in every file into a single dictionary. This makes it particularly useful for large configurations, such as groups, customizations, or templates, where multiple related entries can be organized into separate files.

For example, add the line `group: !include_dir_merge_named group/` to `configuration.yaml` you can organize your groups into multiple files based on rooms, device types, or functions.

Table 2.3 compares how the configuration appears before and after using !

`include_dir_merge_named`.

configuration.yaml	group/switches.yaml
<pre>group: Living Room Lamps: entities: - switch.livingroomlamp1 - switch.livingroomlamp2</pre>	<pre>Living Room Lamps: entities: - switch.livingroomlamp1 - switch.livingroomlamp2</pre>

Table 2.3 – Directive `!include_dir_merge_named` folder/ example

As shown in *Table 2.3*, the `group:` key is defined only once in `configuration.yaml`, while the group definitions are stored in separate files within the `group/` directory. When Home Assistant starts, it reads every file in that directory and merges their contents into a single dictionary. For example, if the `group/` directory contains `switches.yaml`, `lights.yaml`, and `family.yaml`, all group definitions become part of the same `group:` section in the final configuration.

Now that you understand how include directives help organize large configurations, the next section explores additional strategies for structuring your Home Assistant configuration for long-term scalability and maintainability.

Modularizing your configuration: strategies for scalability

The `include` statements we just saw allow you to split and modularize your configuration. This modularization is important in case you grow your home with multiple devices. These multiple devices, helpers, configurations can turn your home difficult to manage in case problems starting to happen with your scenes, scripts or automations for example.

Modularization enables you to isolate your configuration in different files organized in such way you can understand and find them quickly. You can better isolate them to search for potential issues. You can also reuse code from, for example, a room to another by just copy a folder.

We will see two strategies you can use to modularize your configuration allowing scalability.

Strategy A — breaking configuration into domain-specific files

This strategy consists in keeping each domain in its own file or folder. This mirrors Home Assistant's domains or integrations (sensors, automations, scenes, scripts, etc.) and is great for beginners through advanced users. As my devices and entities keep growing I had to implement this strategy to avoid having a huge `configuration.yaml` file. Here is a part of my `configuration.yaml` file:

```
# Domain-centric includes
automation: !include_dir_merge_named automations/
script: !include scripts.yaml
scene: !include scenes.yaml
sensor: !include_dir_merge_list sensors/
group: !include_dir_merge_named group/
template: !include_dir_named template/
mqtt: !include_dir_named mqtt/
shell_command: !include_dir_merge_named shell_command/
```

This uses the same approach Home Assistant already applies by default for scripts and scenes. It will also create separate folders for automations, sensors, groups, templates and mqtt integrations where I have different files. For the group integration, which I use often, I have the following folder tree:

```
group/
├─ door_sensors.yaml
└─ lights.yaml
```

The `door_sensors.yaml` file will contain the following:

```
all_doors:
  name: All doors
  entities:
    - binary_sensor.front_door
    - binary_sensor.kitchen_backyard_door
    - binary_sensor.master_bedroom_backyard_door
    - binary_sensor.garage_door

all_garage_gate:
  name: All garage gate
```

```
entities:
  - binary_sensor.garage_gate_door
  - binary_sensor.z_garage_gate_contact
```

You may prefer to use strategy A if you want to use a native Home Assistant style layout that maps straight integrations in files and folders.

Note

It is a good practice to keep an `automations.yaml` file in the config folder as the automation editor creates automations there and these will be needed at least to create an initial unique numeric id.

Strategy B — organizing by function

Using strategy B, you group everything needed for a feature like security, climate, or entertainment into one folder. Each file can include automations, helpers, templates, and UI customizations for that feature in a portable and self-contained way. The key configuration parameter you can use to create this approach is using packages (<https://www.home-assistant.io/docs/configuration/packages>). You can include for example the directive below in the `configuration.yaml` to create it.

```
homeassistant:
  packages: !include_dir_merge_named packages/
```

You can include in the packages folder a file for each feature as described. In each of these files you can mix different integrations. For example, I created a package to deal with security, where there is a configuration for a manual alarm, a template that counts how many open doors are at home and an automation that sends a message to my phone if a door is opened or closed after the sunset. The stripped code for the file `security.yaml` stored at the folder `packages/security` is like below:

```
alarm_doors_at_night:
  alarm_control_panel:
    - platform: manual
      name: home_alarm
# ...

homeassistant:
  customize:
```



```

    alarm_control_panel.home_alarm:
      icon: mdi:security-home
      friendly_name: "Home Alarm"

  template:
    - sensor:
      - name: "Doors Open Count"
# ...

  automation:
    - id: '1758422161490'
      alias: Notify Door Open at Night
# ...

    data:
      message: "Door open at night: {{ states('group.all_doors') }}"

```

- You can use strategy B if you want to keep certain features like security, climate as portable modules or you maintain multiple homes and want to copy features between them. The good news is that if you like both strategies A and B, you can use both. The trick is defining a clean boundary between both configurations, avoiding situations where they conflict with each other.
- As we learned about these two strategies let's learn some best practices on folder structure and naming convention.

Best practices on folder structure and naming conventions

- Using Strategy A, which is domain-based, keep folder names as close to the integration names as possible, like automation(s), scene(s), script(s), etc. Using strategy B, which is feature-based, try to keep the folder name as close to the feature as possible. Names like security, climate and entertainment are indicated in this case.
- Names can vary according to your choice, but I recommend the following:
- Keep names short, and use prefixes as your setup grows.
- Use lower case, underscores for spaces: living_room, kitchen_lights, water_leak_sensor.
- Be descriptive, but concise: garage_motion_sensor, bathroom_vent_switch.
- Use Entity IDs following the standard {domain/integration}.{area}_{device} or {domain/integration}.{area}_{device}_{attribute}. For example:

```
binary_sensor.kitchen_motion, switch.office_fan or  
sensor.kitchen_fridge_temperature.
```

Try to use names for files and folders for automations and scripts grouped by room or purpose if possible.

The example below shows some suggestions mentioned for folder structure and name convention:

```
Config/  
├─ configuration.yaml  
├─ automations/ # domain files (Strategy A)  
│   ├── living_room.yaml  
│   └── kitchen.yaml  
├─ scenes/  
├─ scripts/  
├─ sensors/  
├─ templates/  
├─ switches/  
└─ packages/ # feature packages (Strategy B)  
    ├── security/  
    │   └── security.yaml  
    ├── climate/  
    │   └── climate.yaml  
    └── entertainment/  
        └── entertainment.yaml
```

Version control with Git: managing your Home Assistant configuration

- As your Home Assistant configuration grows in size and complexity, keeping track of changes in YAML files becomes essential. Without a proper system, you are at the risk of losing working configurations or introducing errors that are difficult to debug. Version control with Git provides a solution to these problems. With Git, every modification you make to your configuration files is recorded as a commit, giving you the ability to track history, roll back to a previous working state, and collaborate with

others. This practice turns your Home Assistant installation into a project managed with the same discipline as a software application.

- My primary use for Git is backing up and testing new configurations, since I'm constantly experimenting with changes to my setup. I'll explain how I back up my Home Assistant config with Git from time to time and also how to retrieve it if needed.

Create a backup routine using GitHub

You have to run the SSH app to access the Home Assistant shell. Assuming you are running a Home Assistant OS installation you have to follow the sequence:

1. Change to the config folder and initialize a new git repository in the current directory:

```
[core-ssh ~]$ cd config
[core-ssh config]$ git init
```

2. Create a .gitignore file and include all files you don't want in the backup like secrets.yaml, database (*.db), log files (*.log), backup files if you have them (.restore folder), etc.
3. Add the files to the commit and commit the files:

```
[core-ssh config]$ git add .
[core-ssh config]$ git commit -m "Initial Home Assistant config"
```

4. Make sure to move from master branch name to main:

```
[core-ssh config]$ git branch -m master main
```

5. As Git handles SSH keys, you need to create your own ssh keys using the following commands:

```
[core-ssh config]$ mkdir .ssh
[core-ssh config]$ ssh-keygen -t rsa -b 4096 -C "your_Github_account"
Enter file in which to save the key (/root/.ssh/id_rsa): .ssh/id_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in .ssh/id_rsa
Your public key has been saved in .ssh/id_rsa.pub
```

6. Copy and add the SSH key generated from file ssh/id_rsa.pub to the SSH and GPG Key section at <https://github.com/settings/ssh/new>

7. Enter the following command to inform the `id_rsa` file location.

```
[core-ssh config]$ git config core.sshCommand "ssh -i /config/.ssh/id_rsa -o 'StrictHostKeyChecking=no' -F /dev/null"
```

8. Create a GitHub repository where you want to store your Home Assistant configuration and use the following commands to push the files to your remote repository in GitHub:

```
[core-ssh config]$ git remote add origin  
git@github.com:your_Github_account/your_remote_git_repository.git  
[core-ssh config]$ git push -u origin main
```

9. At this point, your files in the `config` folder will be pushed to `your_remote_git_repository`. Let's create now a script to backup the `config` folder from time to time to your git repository. Create a file called `backup_github.sh` with following content:

```
HA_VERSION=`cat .HA_VERSION`  
COMMIT_DATE=$(date +%d-%m-%Y %H:%M:%S)  
COMMIT_MESSAGE="Home Assistant Commit - [$HA_VERSION]: $COMMIT_DATE"  
  
echo "$COMMIT_MESSAGE"  
  
git add .  
git commit -m "$COMMIT_MESSAGE"  
git push
```

10. Set executable permissions to the script just created and run the script to see results and check for errors:

```
[core-ssh config]$ chmod +x backup_github.sh  
[core-ssh config]$ ./backup_github.sh
```

11. If you get no errors, your script works and now you can automate the backup to github. First create a service to run the script created by adding the following lines to your configuration.yaml

```
shell_command:
  run_githubbackup: bash ./backup_github.sh
```

12. Create an automation depending on what frequency you want your data backed up. I run mine daily at 23:59. If there is no change in your config folder, nothing will be pushed. My automation looks like the following:

```
- id: '1758603098571'
  alias: Run Daily Config Github backup
  description: ''
  triggers:
    - trigger: time
      at: '23:59:00'
  conditions: []
  actions:
    - action: shell_command.run_githubbackup
      data: {}
  mode: single
```

Now, you have a Home Assistant Config backup being done using GitHub. Next section, we will see how to retrieve files from GitHub.

Retrieve backup files using GitHub

I developed a set of artifacts to retrieve a single file from GitHub. The idea came from situations where I was debugging YAML code, accidentally lost a file completely, and needed to pull it back into my Home Assistant installation. Of course, you can retrieve this file just using regular GitHub commands in the SSH terminal from Home Assistant but the goal here is to demonstrate its capabilities to run shell commands, shell scripts, helpers and scripts coordinated. We will split our development in four steps:

- Shell command and shell script development
- Helper development
- Script development
- Dashboard integration

Shell command and shell script development

We will start developing a shell script that takes five parameters from the user and use these parameters to pull the desired file from GitHub. These parameters are:

1. Owner – your GitHub username
2. Repo – The repository name you want the file to be pulled from
3. Ref – which branch you want to pull the file from
4. Path – Path and filename where you want to pull from GitHub
5. Dest – Destination in your local installation you want to store the file

The shell command file will be stored in the `shell_command` folder and we will name it as `github_fetch_file.yaml`. This file will have the following content.

```
github_fetch_file: >
  bash -lc
  "/config/shell_scripts/github_fetch.sh '{{ owner }}' '{{ repo }}' '{{ ref }}'
  '{{ path }}' '/config/{{ dest }}'"
```

The shell command just ensures that the script `github_fetch.sh` will run as a single command from a bash shell and using the five parameters we just commented.

The shell script should take these five parameters and format a command to be invoked by a curl command which uses a GitHub API syntax. Before we provide the shell script code, to pull the file from GitHub we will have to create a new fine-grained token using the GitHub website (<https://github.com/settings/personal-access-tokens>). Click on Generate new token button and configure the name, expiration, repository access and permissions. Click Generate token and copy it to a file to be created in your Home Assistant installation. I named the file as `github_token` and stored it in the `.secrets` folder.

The shell script will look like the following:

```
#!/usr/bin/env bash
set -Eeu

OWNER="${1:-}"; REPO="${2:-}"; REF="${3:-main}"; FILEPATH="${4:-}"; DEST="${5:-}"
: "${OWNER:?missing OWNER}"; : "${REPO:?missing REPO}"; : "${FILEPATH:?missing
PATH}"; : "${DEST:?missing DEST}"

TOKEN_FILE="/config/.secrets/github_token"
TOKEN="$(cat "$TOKEN_FILE")"

URL="https://api.github.com/repos/${OWNER}/${REPO}/contents/${FILEPATH}?ref=${
REF}"
curl --fail --ssl \
  -H "Authorization: Bearer ${TOKEN}" \
  -H "Accept: application/vnd.github.v3.raw" \
  "$URL" -o "$DEST"

echo "Saved ${OWNER}/${REPO}@${REF}:${FILEPATH} -> ${DEST}"
```

The script reads the five parameters and stores them in the OWNER, REPO, REF, FILEPATH and DEST variables. It also reads the content of the github_token file to the TOKEN variable. The URL variable is then formatted using the parameters OWNER, REPO, FILEPATH, REF and the curl command groups everything together, providing a token and the destination location using the DEST variable.

After you reboot your system, you will be able to test your github_fetch_file shell command and script using the sequence Developer Tools | Actions in Home Assistant, providing the parameters as shown in *Figure 2.2*. I used the file scenes.yaml to be retrieved from GitHub.

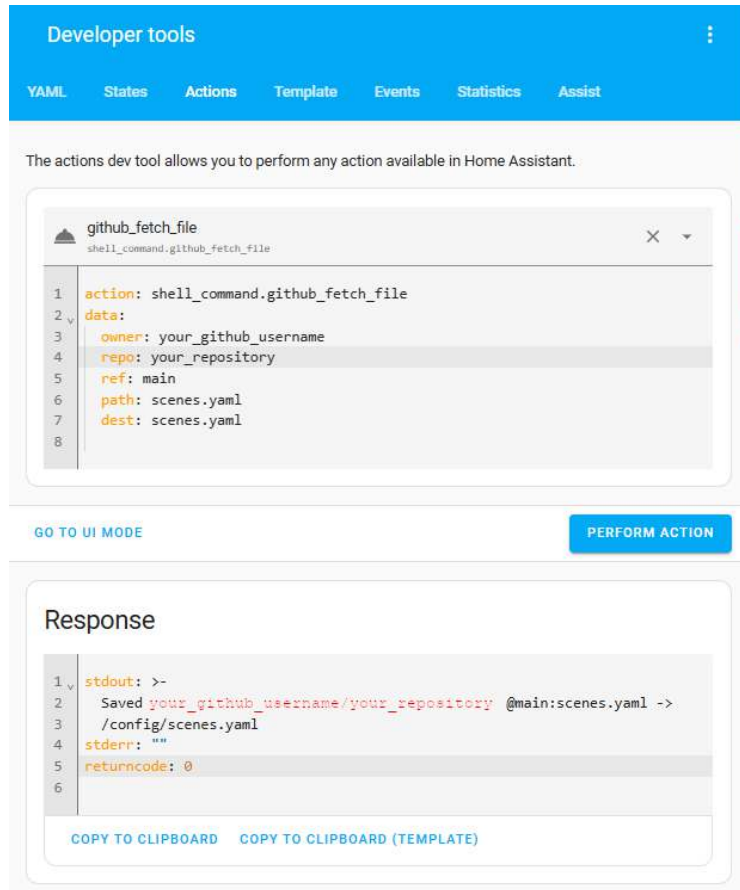


Figure 2.2 – GitHub_fetch_file shell command test

If everything goes well you will have in the Response box the return variables `stderr: ""` and `returncode: 0` as outputs and your `scenes.yaml` will be pulled from GitHub. In the next subsection, we will create the helpers to input some parameters to the script we will create just after.

Helper development

We will create three helpers that will support us input some of the five parameters mentioned before: Ref, Path and Dest. We will create Ref and Dest as input text and Path as input select type. You will create the helpers following the sequence:

1. Go to **Settings | Devices & services | Helpers**.

2. Click on the + **Create helper** button. Select **Text** and in the **Create Text box** fill the **Name*** field as **GitHub reference (branch/tag/sha)**. This will be the Ref parameter.
3. Do the same as number 2 and create another **Text** input and in the **Name*** field include **GitHub Destination path under /config**. This will be the Dest parameter.
4. To create the input select helper, click on + **Create helper** button and then in **Dropdown**. In the **Create Dropdown box** fill out the **Name*** field as **GitHub File Selection**. This will be the Path parameter.
5. In the Add option field add as many files you want to retrieve from GitHub. Click on the + **Add** button to add the next filename. Click on **Create** button after you add all the filenames you need from GitHub. After created, the helper will look like presented in *Figure 2.3*.

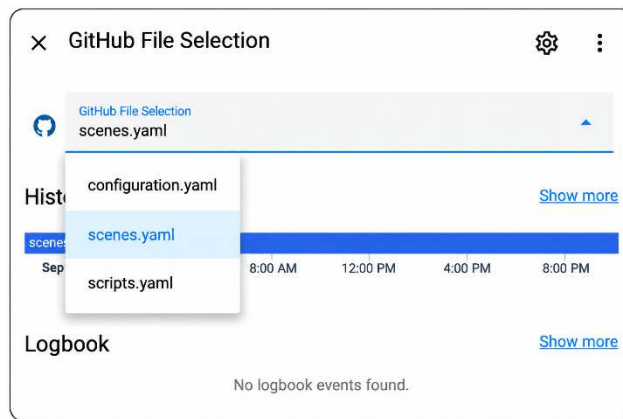


Figure 2.3 – GitHub file selection helper

You now have three helpers to support the script's input information. We will create the script in the next subsection.

Script development

The script groups the five parameters and calls the shell command we saw in the previous section. The first two parameters, owner and repo will be entered as part of the data structure of the script. The other remaining parameters, path, ref, dest will be obtained from the helpers created in the previous section. The main script code could be implemented as following:

```
alias: Fetch selected file from GitHub
mode: single
fields:
```

```
owner:
  description: GitHub owner/org
  example: your_github_username
repo:
  description: GitHub repository name
  example: your_repository
sequence:
  - variables:
      _path: "{{ states('input_select.github_file_to_fetch') }}"
      _ref: >-
        {{ (states('input_text.github_ref') | trim) if
        (states('input_text.github_ref') | trim) else 'main' }}
      _dest: >- {{ (states('input_text.github_dest') | trim) if
        (states('input_text.github_dest') | trim) else
        states('input_select.github_file_to_fetch') }}
  - data:
      owner: "{{ owner }}"
      repo: "{{ repo }}"
      ref: "{{ _ref }}"
      path: "{{ _path }}"
      dest: "{{ _dest }}"
      action: shell_command.github_fetch_file
```

You can test your script in the same way you tested the Shell command using Developer tools | Actions. Select the Fetch selected GitHub Script. Make sure to switch to YAML mode and include the information for owner and repo. Also go to Settings | Devices & Services | Helpers and configure the path, ref and dest parameters. Click on the PERFORM ACTION button.

Figure 2.4 shows how the script configuration will look like.

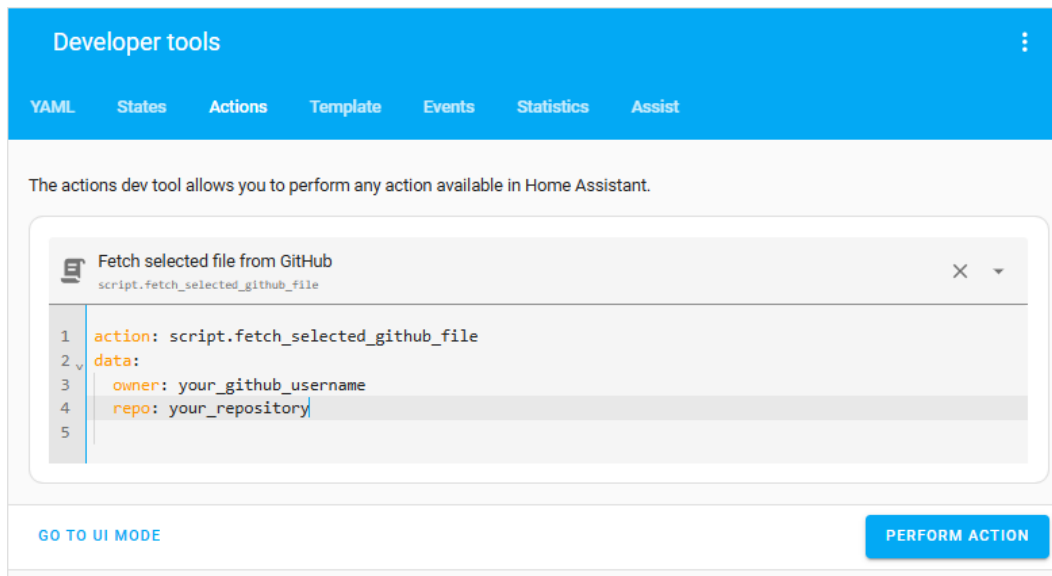


Figure 2.4 – Script fetch selected GitHub file configuration

Your script will be executed and your selected file will be restored. If you want some output for the script command you can add to the script a notification that will be displayed in the notifications tab in the side bar. You should add the following YAML code to your script:

```

- data:
  title: GitHub Fetch
  message: >-
    Pulled {{ _path }} @ {{ _ref }} into /config/{{ _dest }} from {{ owner }} /
    {{ repo }}.
  action: persistent_notification.create

```

You will then have the notification in *Figure 2.5*.

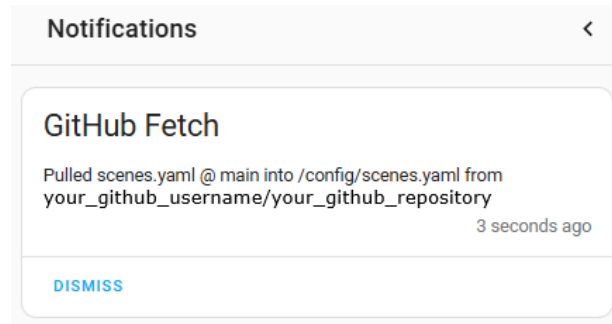


Figure 2.5 – Notification sent after script execution

Dashboard integration

For the dashboard integration, we will include all three helpers using an entity card, and a button card attached to the script developed in the previous section. The button card configuration should have the Interactions fields configured according to the *Figure 2.6*.

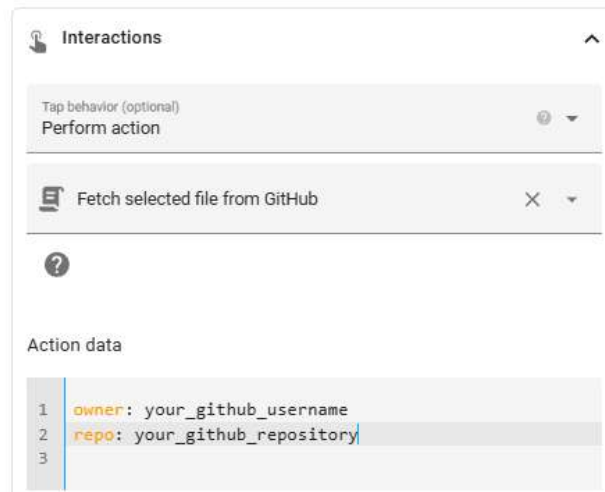


Figure 2.6 – Button card configuration – script

After you configure all four cards, you will have a dashboard configuration like *Figure 2.7*. If you configure your helpers Filepath, Destination and Branch/Tag/Sha like in the *Figure 2.7*, you will have the file `scenes.yaml` be pulled from `your_github_repository` after you click on the Fetch Files from GitHub button.

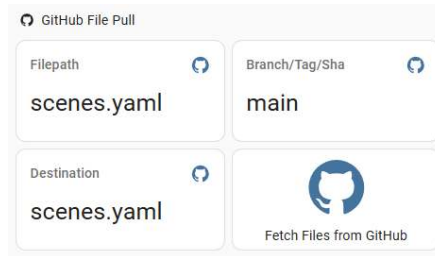


Figure 2.7 – Dashboard configuration – pull files from GitHub

We have now completed our tool to fetch files from GitHub. Next section we will know more about how to store secret or confidential information in Home Assistant.

Secrets management and sensitive data handling

When working with Home Assistant, it's common to deal with credentials, API keys, Wi-Fi passwords, and access tokens. Storing these directly inside `configuration.yaml` or automation files is not only insecure but also makes your setup harder to maintain, especially if you use version control like GitHub. In this section, we'll cover best practices for isolating sensitive data, using the built-in `secrets.yaml` file.

Using `secrets.yaml` in Home Assistant

Home Assistant provides a native way to store and reference sensitive data via `secrets.yaml`. You can create a `secrets.yaml` and store it in the `/config` folder. In this file, include all sensitive information such as usernames, passwords, passkeys, tokens, etc. An example of the content of this file is shown as follows.

```
mqtt_user: your_mqtt_user
mqtt_password: your_password
```

Inside any YAML file, replace the actual value with `!secret <name>`. In this case the mqtt secret information can be referenced in a YAML file as the following example:

```
mqtt:
  broker: 192.168.1.100
  username: !secret mqtt_user
  password: !secret mqtt_password
```

Using this approach with the `secrets.yaml` file, you can better manage your credentials by placing them in a centralized location. You can also prevent external access to your keys and

passwords if you're using GitHub, by configuring your `.gitignore` file to exclude `secrets.yaml` from any commit or push. Keeping sensitive information in one file also makes it easier to update, instead of editing it across multiple files.

We will finish this chapter by discussing troubleshooting YAML code in the next section.

Troubleshooting YAML: common pitfalls and debugging techniques

YAML's simplicity is one of its major strengths, but it also introduces errors that can lead to validation messages or failed system startups. In this section, we will learn how Home Assistant interprets YAML, explore the most common issues that occur, and develop an approach to debug and prevent them. We will also review best practices for working safely with YAML.

Understanding YAML parsing in Home Assistant

Before you can effectively debug or structure your configuration, it's important to understand what happens behind the scenes when Home Assistant reads your YAML files. This process directly impacts how errors appear and why a small mistake can prevent your system from starting correctly.

When Home Assistant starts, it reads your configuration files and merges them into a single unified structure before validating each section. It uses the Python PyYAML library to parse this data into objects that each integration can understand and process.

The process can be summarized as follows:

1. All referenced files and folders are merged using directives like `!include` or `!include_dir_merge_named`.
2. The YAML structure is parsed and converted into internal Python objects.
3. Each integration applies its schema validation, ensuring data types and required fields are correct.
4. If the configuration passes validation, entities are created and registered in the system.

Understanding this sequence is important because a single syntax error can break an entire section of your configuration, preventing Home Assistant from loading correctly.

Consider the example below:

```
mqtt:
  sensor:
    state_topic: "tele/ESP32_TEMPERATURE_HUB/SENSOR"
```

```
    unit_of_measurement: "°C"
    name: "Backyard Temperature"
    value_template: '{{ value_json.ATC19f133.Temperature }}'
```

In this example, the indentation of the last line is off by one space. Because YAML relies strictly on indentation to define structure, this small error causes Home Assistant to fail when parsing the mqtt section.

This highlights the important principle that in YAML, structure is defined by spacing, not symbols. Even minor inconsistencies can lead to parsing failures, making careful formatting and validation essential when working with complex configurations.

Common YAML errors and how to fix them

Table 2.4 summarizes the most frequent YAML issues in Home Assistant configurations.

Error Type	Incorrect Example	Root Cause	Fix
Indentation Error	automation: - alias: Light On trigger: platform: state	Misaligned spaces or missing hyphen	Always use 2 spaces per level, never tabs
Improper Quoting	name: John's Room	Apostrophe terminates string unexpectedly	Use double quotes → name: "John's Room"
List vs Dictionary Mix	sensor: name: Temp - platform: mqtt	Mixing key/value pairs with list entries	sensor: - platform: mqtt name: Temp
Wrong Data Type	delay: "10"	Value interpreted as string instead of integer	delay: 10

Error Type	Incorrect Example	Root Cause	Fix
Unrecognized Integration Key	homeassistnat: misspelled	Invalid root key name	Check spelling and use the Configuration Checker

Table 2.4 – YAML issues in Home Assistant configurations

Note

Whether a value should be a string, integer, Boolean, or list depends on the integration's schema. Always consult the integration documentation if you're unsure of the expected data type.

In Home Assistant, these issues are typically reported by debug tools. If you understand the terminology of their messages, you can find out where the structure breaks. We will see in the next subsection some of these debug tools used to find these errors.

Using built-in debugging tools

Home Assistant provides some tools to help validate and debug YAML configurations before you restart your system. Use these tools to reduce the risk of mistakes in the configuration.

I use mostly the configuration validation tool. From the web interface, navigate to **Developer tools | YAML | Check and restart**. Click on **Check Configuration**. If everything is fine in your configuration, a green message saying, "Configuration will not prevent Home Assistant from starting!" will be presented, otherwise an error or warning message will be displayed.

You can also use the following command line to check the configuration. The command will parse and validate all configuration files and if everything is OK, the successful message will be presented as in the following command output.

```
[core-ssh config]$ ha core check
Processing... Done.

Command completed successfully.
```

An external tool that I use is the VS-code App extension. It provides YAML syntax highlighting and schema validation.

We will explore another tool in the next chapter to support the development of templates and sensors verification. For now, using Developer Tools | Template tab allows you to test and

debug templates directly. This helps you to confirm whether your syntax is correct before embedding it in YAML code.

Preventive best practices

Debugging YAML becomes much easier when configurations are structured properly from the beginning. The following guidelines can help you avoid many pitfalls:

1. Use two spaces per indentation level, and never use tabs.
2. Keep your configuration modular using `!include_dir_merge_named` or similar directives.
3. Comment complex sections extensively.
4. Validate configurations after every significant change.
5. Keep version control with GitHub to easily roll back if something breaks.
6. Test all Jinja2 templates in the Template Editor before saving.
7. Maintain a minimal, working configuration backup to recover quickly.

With these guidelines, we finish this second chapter. Next chapter we will discuss more about Templates development.

Summary

In this chapter, you learned how to master YAML for managing complex Home Assistant configurations. It started with advanced YAML features such as anchors, aliases, and includes, showing how they help reduce repeated code and keep configurations organized. You also learned how to split and structure configuration files to make them easier to maintain and expand.

You explored two main strategies for organizing configurations, by domain and by function, so your setup could grow without becoming difficult to manage. The chapter also covered how to use Git for version control, including creating scripts and automations to back up and retrieve files from GitHub.

Finally, you learned how to protect sensitive information using the `secrets.yaml` file and how to troubleshoot common YAML errors. You practiced using Home Assistant's built-in validation and debugging tools and reviewed best practices to prevent future configuration issues.

In *Chapter 3, Advanced Templating in Home Assistant*, you'll build on your YAML configuration and structuring skills by exploring Jinja2 templates, enabling you to create dynamic and data-driven automations that make your Home Assistant installation more intelligent and responsive.

3

Advanced Templating in Home Assistant

Templates are what transforms Home Assistant from a system that simply reacts to events into one that can interpret data and make intelligent decisions. In this context, templating is powered by Jinja2, which allows you to process entity states, evaluate conditions, and dynamically generate values used across automations, sensors, and the user interface. This capability becomes essential when you need to go beyond static configurations and build logic that adapts to real-time conditions. Building on the YAML foundations covered in the previous chapter, this is the next step in creating more flexible, scalable, and maintainable automation systems.

In this chapter, you will learn how to use templates to extract and manipulate data, create dynamic sensors and automations, and test your logic using built-in tools. By the end of this chapter, you will be able to design data-driven automations that respond intelligently to changing conditions and improve the overall behavior of your Home Assistant system.

The following topics will be explained in this chapter:

- Jinja2 fundamentals in Home Assistant: variables, filters, and tests
- Dynamic data extraction and manipulation
- Advanced templating: conditionals, loops and complex logic
- Using the Home Assistant Template Editor for debugging and testing
- Real-world application: dynamic notifications, sensors, and UI customization

Note

Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Technical requirements

To get the most out of this chapter, you should have the following knowledge and setup:

- A running Home Assistant installation with access to the Developer Tools and Template Editor
- Basic familiarity with editing Home Assistant YAML configuration files using Visual Studio Code, File Editor, or another text editor
- A basic understanding of Home Assistant entities, including sensors, binary sensors, states, and attributes
- Familiarity with Home Assistant automations, helpers, and dashboard cards
- Basic programming concepts such as variables, comparison operators, conditional statements (`if/elif/else`), loops, and functions. Previous experience with Python is helpful because Jinja2 uses a Python-like syntax, but no prior Python programming experience is required.
- If you need a refresher on Home Assistant templating, refer to the official Home Assistant Templating documentation: <https://www.home-assistant.io/docs/configuration/templating/>
- If you are unfamiliar with Home Assistant entities and states, refer to the official documentation: https://www.home-assistant.io/docs/configuration/state_object/

Jinja2 fundamentals in Home Assistant: variables, filters, and tests

Home Assistant is powerful not just because it supports many integrations and automations, but because it can handle and display data in flexible ways. Templating, especially with Jinja2, is what turns fixed configurations into smart, data-driven automations.

With templates, you can make your system react to real situations. For example, show messages based on current sensor readings, calculate values instantly, or decide what actions to take depending on multiple conditions. Instead of using fixed numbers or text, templates let Home Assistant make decisions based on live data from entity states, attributes, and events.

Jinja2 templating appears throughout Home Assistant from automations, scripts, sensors, and notifications, to cards in dashboards. While YAML defines the structure and configuration, Jinja2 acts as the "brain" of the system, retrieving data, evaluating conditions, making decisions, and generating dynamic content based on real-time information. It transforms static YAML configurations into intelligent, responsive automations. In the next subsections, you will learn the fundamentals of Jinja2, including its syntax, variables, filters, and tests.

Understanding Jinja2 syntax and integration with Home Assistant

Jinja2 is a Python-based templating language. In Home Assistant, it is tightly integrated with the core system, allowing direct access to entity states, attributes, and services. As introduced in *Chapter 1*, Home Assistant's state machine maintains the current state of every entity, while the service registry provides access to the actions that entities and integrations can perform. This integration enables templates to retrieve real-time information, evaluate conditions, and generate dynamic content throughout the system.

The syntax is simple yet powerful. Some syntax examples are shown in *Table 3.1*:

Jinja2 Syntax	Description	Example
<pre>{{ .. . }}}</pre>	Outputs a value (expression or variable)	<pre>{{ states('sensor. kitchen_temperature') }}</pre>
<pre>{% .. . %}</pre>	Executes logic (loops, conditionals, etc.)	<pre>{% if is_state('light.kitchen', 'on') %} Light is ON{% endif %}</pre>
<pre>{# ... #}</pre>	Comment — ignored in output	<pre>{# This is a comment #}</pre>

Table 3.1 – Jinja2 syntax examples

By using Jinja2, Home Assistant automatically makes many helper functions, objects, and status data available inside templates. For instance, the functions `now()` and `today_at()` work with timestamps and scheduling logic, respectively. These quick example functions show how Jinja2 can be a powerful tool used to automate and output information in Home Assistant. More information about Jinja2 can be found here: <https://jinja.palletsprojects.com/en/stable/>

Working with variables

Variables in Jinja2 work like any other variable found in other programming languages. You can define and use variables inside templates using the `set` keyword. The code below gives us an example of the use of variables.

```
{% set temperature = states('sensor.kitchen_temperature') %}
```

In this example, the variable `temperature` is assigned a value read directly from a Home Assistant sensor. Variables can be numeric, string, boolean, or even complex data like lists and dictionaries.

Variables can be used, for example, to store intermediate calculations (e.g., average temperature), to cache repeated calls to the same entity or to simplify complex automation logic for readability.

Using filters

Filters are used to modify or format data. They are applied using the `|` (pipe) operator, similar to how Unix pipes work. An example of how filters are used is shown in the following code:

```
{{ states('sensor.kitchen_temperature') | float | round(2) }}
```

The sensor value in the code shown before is first retrieved as a string, then converted to a floating-point number and rounded to two decimal places. As seen in the example, filters can be chained together, enabling you to manipulate data step by step.

Table 3.2 shows common Jinja2 filters used in Home Assistant.

Filter	Description	Example	Result
<code>float</code>	Converts string to floating number	<code>{{ '12.7' float }}</code>	<code>12.7</code>
<code>int</code>	Converts string to integer	<code>{{ '5.9' int }}</code>	<code>5</code>
<code>round(n)</code>	Rounds number to n decimals	<code>{{ 3.14159 round(2) }}</code>	<code>3.14</code>
<code>capitalize</code>	Capitalizes first letter	<code>{{ 'living room' capitalize }}</code>	<code>Living room</code>
<code>lower / upper</code>	Changes text case	<code>{{ 'HOME' lower }}</code>	<code>home</code>
<code>replace(a, b)</code>	Replaces text	<code>{{ 'fan_on' replace('_', ' ') }}</code>	<code>fan on</code>

Filter	Descripti on	Example	Result
<code>join(', ')</code>	Joins list items	<code>`{{ ['TV', 'Lamp']   join(', ') }}`</code>	<code>`TV, Lamp`</code>
<code>length</code>	Returns list length	<code>`{{ states.light   length }}`</code>	<code>`9`</code>
<code>default(value)</code>	Uses default if variable is undefined	<code>`{{ value   default('unknown') }}`</code>	<code>`unknown`</code>
<code>timestamp_custom('%H: %M')</code>	Formats timestamps (current example hour 13:32)	<code>`{{ now()   as_timestamp   timestamp_custom('%H: %M') }}`</code>	<code>`13: 32`</code>

Table 3.2 – Common Jinja2 filters

Using tests

Tests are similar to filters but return a boolean value (True or False). They are used in conditional expressions to evaluate logic. Table 3.3 shows common Jinja2 tests in Home Assistant.

Test	Description	Example	Returns
<code>is number</code>	Checks if value is numeric	<code>`{{ 5 is number }}`</code>	<code>True</code>

Test	Description	Example	Returns
<code>is string</code>	Checks if value is text	<code>{{ 'fan' is string }}</code>	True
<code>is defined</code>	Checks if variable exists	<code>{{ my_var is defined }}</code>	False (in my setup)
<code>is iterable</code>	Checks if value is iterable (list, dict)	<code>{{ [1,2,3] is iterable }}</code>	True
<code>is divisibleby(n)</code>	Checks divisibility	<code>{{ 10 is divisibleby(5) }}</code>	True
<code>is even / is odd</code>	Checks if number is even/odd	<code>{{ 3 is odd }}</code>	True
<code>is sameas(value)</code>	Compares identity	<code>{{ a is sameas(b) }}</code>	True/ False

Table 3.3 – Common Jinja2 tests

Understanding these fundamentals will prepare you for the next section, where we will use these tools to extract and manipulate real-time data from Home Assistant's states, attributes, and services enabling the implementation of more advanced automations.

Dynamic data extraction and manipulation

Templating is how you turn raw entity data into decisions, numbers, and messages your automations can use. In Home Assistant, you'll mostly pull information from:

- States (e.g., on, 23.7, unavailable)

- Attributes (metadata on an entity, e.g., `unit_of_measurement`, `battery_level`)
- Collections of entities (e.g., groups, areas, device classes)

This section shows how to extract and process Home Assistant data in real time using Jinja2 functions and filters such as `expand()`, `selectattr()`, `rejectattr()`, `map()`, `sum`, `max`, `min`, and others. You will build practical templates and then incorporate them into template sensors so the values update efficiently. For example, later in this chapter you will use these techniques to calculate the operating cost of a garage refrigerator based on its energy consumption, generate threshold-based notifications, and display the results on a dynamic Home Assistant dashboard.

Getting data: states, attributes, and groups

We can get data from entities using basically four core helper functions:

- `states('entity_id')` - returns the state as a string. Returns 'unknown' if missing.
- `State_attr('entity_id', 'attr')` - returns the attribute of an entity. Returns None if missing.
- `Is_state('entity_id', 'value')` - returns true or false depending if `entity_id` state is equal to 'value'.
- `expand('group.name', 'entity_id', ...)` - returns a list of State objects for those entities.

The following examples show the return values of the functions described earlier:

```
{{ states('sensor.kitchen_temperature') }}
{# returns: "22.3" #}

{{ state_attr('sensor.kitchen_temperature', 'unit_of_measurement') }} {# returns:
"°C" #}

{{ is_state('binary_sensor.front_door', 'on') }} {# returns: true/false #}

{{ expand('group.all_doors') | list | length }}
{# returns: number of entities in the group #}
```

Selecting and transforming entity lists

When you use the `expand()` function to expand a group, you get a list of State objects where each one has `.entity_id`, `.state`, `.attributes`. You can then use attribute filters to slice and transform. Some functions can be used in this case:

- `selectattr('state', 'eq', 'on')`: keep only the attributes where condition matches. In this case this filter will keep the entities where the state 'state' should be equal ('eq') to 'on'.
- `rejectattr('attributes.device_class', 'ne', 'door')`: opposite from previous function. It will drop entities where the device class attribute is not equal ('ne') to 'door'.
- `map(attribute='entity_id')`: Extract a field from each entity. In this case the 'entity_id' attribute will be pulled from the group for each entity
- `list`: convert entity attributes in a list.

The following example shows how we can use the filter functions mentioned. I used the following example as a template sensor to count and list the doors open at my house. This is part of my security package demonstrated in *Chapter 2*.

```
template:
- sensor:
  - name: "Doors Open List and Count"
    state: >
      {# Get all door entities #}
      {% set doors = expand('group.all_doors') | list %}

      {# Keep only open doors #}
      {% set open_doors = doors | selectattr('state','eq','on') | list %}

      {# Show count and list #}
      Open doors ({{ open_doors|length }}):
      {{ open_doors | map(attribute='entity_id') | list }}
```

When some door is opened in the house the sensor produces the following example output:

```
Open doors (1):  
[ 'binary_sensor.kitchen_backyard_door' ]
```

Numeric calculations (sum, min, max)

Using the functions we saw in the previous subsection and also others like sum, average, min and max you can play with some calculations or numeric manipulations with your devices and entities.

In this section we will use data from the temperature sensors I have in my house. I have 3 temperature sensors on the north side of my house, so I created a sensor using the Template integration in Home Assistant. This template sensor combines the average temperature of these three temperature sensors installed on the north side of the house. I also created other template sensors to combine the average temperature from the temperature sensors installed on the south side and also the entire house temperature internally. The template integration code for north temperature average sensor uses the sum function and is as follows:

```
template:  
  - sensor:  
    - name: "North Temperature Average"  
      unit_of_measurement: "°C"  
      state: >  
        {# Get all north sensors #}  
        {% set rooms = expand('group.all_north_temperature') %}  
  
        {# Convert states to numbers #}  
        {% set values = rooms | map(attribute='state') | map('float',  
default=0) | list %}  
  
        {# Count valid readings #}  
        {% set count = values | select('ne', 0) | list | length %}  
  
        {# Compute average #}  
        {{ (values | sum / (count if count > 0 else 1)) | round(1) }}
```

The other two template temperature sensors were created with the same code above but just replacing the north group of sensors by south and Internal to get the average temperature from south and entire internal temperature of the house respectively.

In the previous code other two functions are also used. They are `select` and `length`. The `select` function applies the condition not equal (`'ne'`) to zero to select the values. The `length` function gets the list size of the temperature values.

You can use the function names `max` and `min` to find the maximum and minimum temperature among the north, south or entire house. You have to replace the last two lines of the previous template code by the code below:

```
{{ values | max }} {# for max temperature #}
{{ values | min }} {# for min temperature #}
```

The template sensor's code can be optimized to just log data when triggered. To do that, you have to remember the section Database maintenance – Filtering undesired information. Basically, you have to replace the group entity with the list of your sensors so the template will only be updated or executed when a value on these sensors changes. In the case of Doors Open List and Count sensor the new code will be the following:

```
template:
- trigger:
  {# Monitor door state changes #}
- platform: state
  entity_id:
    - binary_sensor.front_door
    - binary_sensor.kitchen_backyard_door
    - binary_sensor.garage_door

sensor:
- name: "Doors Open List and Count"
  state: >
    {# Get open doors only #}
    {% set open_doors = expand(
      'binary_sensor.front_door',
      'binary_sensor.kitchen_backyard_door',
      'binary_sensor.garage_door'
    ) | selectattr('state','eq','on') | list %}

    {# Show count and List #}
    Open doors ({{ open_doors|length }}):
    {{ open_doors | map(attribute='entity_id') | list }}
```

Using the knowledge from this section about dynamic data extraction, you can now turn raw entity data into numbers, lists, and summaries that make your automations more dynamic. The next step is combining this data with logic allowing your templates to decide what to do based on conditions, loops, and context. We will see how to do that in the next section.

Advanced templating: conditionals, loops and complex logic

In this section you will learn how to use Jinja2 control structures like for, if/elif/else, and a few other patterns to make decisions from multiple signals from different devices and entities. We will build automations that react to combinations of states, not just a single trigger.

We will start by explaining how to implement conditions in Templates.

Conditionals: indirect implementation and if, elif, else

Like any other programming language, conditional statements help to execute commands or actions if conditions are met (if statement), not met (else statement) or not met initially but met subsequently (elif statement). You can use these conditional statements with Templates. Let's implement two examples involving conditional statements.

Conditional - indirect implementation

The first example turns on the thermostat when the temperature is above a certain value, but with two conditions to be satisfied. The first condition is that all doors are closed, and the second condition is that someone is home. An indirect implementation of conditionals is using templates to define variables and test these variables as condition to execute automations. Check the following implementation:

```
alias: Thermostat smart gate

# Trigger when indoor temperature exceeds 25°C
triggers:
  - trigger: numeric_state
    entity_id:
      - sensor.internal_temperature_average
    above: 25

conditions: []

actions:
  # Store values used later in the automation
  - variables:
```

```
door_open: "{{ is_state('group.all_door','on') }}"
at_home: "{{ is_state('person.your_name', 'home') }}"

# Evaluate the current conditions
- choose:
  # If a door is open, notify the user
  - conditions:
    - condition: template
      value_template: "{{ door_open }}"
    sequence:
      - action: notify.mobile_app_iphone_381
        data:
          message: "Some door is open. Thermostat not started."

  # Otherwise, if someone is home, start the thermostat
  - conditions:
    - condition: template
      value_template: "{{ at_home }}"
    sequence:
      - action: climate.set_temperature
        target:
          entity_id: climate.nest_south
        data:
          temperature: 23

# Otherwise, just Log the event
default:
  - action: logbook.log
    data:
      name: Thermostat smart gate
      message: "No one home. No action done."

mode: single
```

Templates in this automation define boolean variables based on values from two entity states. The automation triggers if the average temperature of my home is above 25C. If it is above 25C, the first boolean variable `door_open` is checked first. If it is true, a message is sent to my phone with the message "Some door is open. Thermostat not started". If all doors are closed then the second variable `at_home` is tested. If I'm at home, the thermostat `climate.nest_south` is set to 23C. If none of the conditions could be met, meaning all doors are closed but nobody is at home, there is a default condition just to log the information that the automation was triggered but no action was effectively done. This arrangement with conditions looks similar to case statements in the C programming language. It is an elegant way to test conditions instead of using nested if/else statements which we will see next.

Nested if/elif/else

The if/elif/else statements follow the general pseudocode syntax:

```
{% if condition_1_met %}  
    Display message_1 or execute action_1  
{% elif condition_2_met %}  
    Display message_2 or execute action_2  
{% elif condition_3_met %}  
    Display message_3 or execute action_3  
{% else %}  
    Display message_4 or execute action_4  
{% endif %}
```

The second example using conditionals will use nested if/elif/else statements to test different temperature thresholds and advise the user for different actions that could be done. The template code reads the temperature from the sensor located in my son's bedroom and present messages to the user. The code that does this is presented in the sequence:


```
{% set t = states('sensor.kid1_room_temperature')|float(0) %}
{% if t >= 30 %}
    It's very hot ({{ t }}°C). Turning on the fan to max.
{% elif t >= 25 %}
    Warm ({{ t }}°C). Fan to medium.
{% elif t >= 21 %}
    Comfortable ({{ t }}°C). Keeping current setting.
{% else %}
    Cool ({{ t }}°C). Turning fan off.
{% endif %}
```

You can use nested if/elif/else statements in template messages. These messages can be used in notification messages, Display cards in dashboards, automations and Template sensor states.

Loops: using for ... in

You can use loops to sweep a list of sensor data and verify for information some specific information. Home Assistant uses Jinja2 that offers the `for ... in` statement to do that. The general format is according to the following pseudocode.

```
{% for each_item in a_list %}
    Do_something_with_this_item
{% endfor %}
```

We will create a practical example from a template code that will filter my battery powered sensors but only the ones used in doors and motion. This sensor information then will be verified using a loop to check if the battery level is low or less than 5%. This information will then be listed. *Figure 3.1* include the flowchart of the example we are describing.

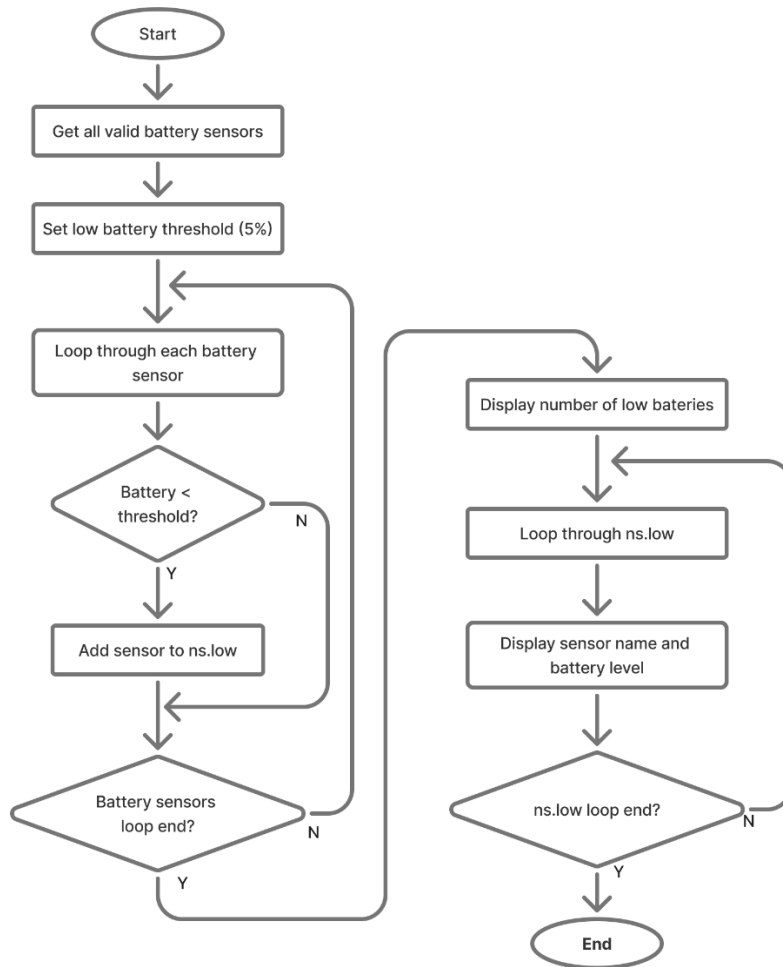


Figure 3.1 – Example using loops in templates

Check the code below to see how we can do it.

```

{# Get valid battery sensors #}
{% set batt_sensor = states.sensor
  | selectattr('attributes.device_class','eq','battery')
  | selectattr('attributes.state_class','eq','measurement')
  | rejectattr('state','in',['unknown','unavailable'])
  | rejectattr('attributes.icon','defined')
  | list %}

```

```

{# Set Low battery threshold #}
{% set low_battery_level = 5 %}

{# Create List for Low batteries #}
{% set ns = namespace(low=[]) %}

{# Find batteries below threshold #}
{% for i in batt_sensor %}
    {% if i.state | float(0) < low_battery_level %}
        {% set ns.low = ns.low + [i] %}
    {% endif %}
{% endfor %}

{# Display summary #}
Low battery entities < {{ low_battery_level }}%: {{ ns.low | length }}

{# Display each Low battery sensor #}
{% for e in ns.low %}
    {{ e.name }} ({{ e.entity_id }}): {{ e.state | float(0) | round(0) }}%
{% endfor %}

```

You can see in the code that the `for ... in` statement is used twice. In the first iteration the dataset `ns.low` is created with the battery sensors which the battery level lower than the `low_battery_level`. You will have to use the `namespace` directive to create the variable `ns` because Jinja2 doesn't persist the data in for loop if you don't do that. The second `for ... in` iteration is used to print the data out of the `ns.low` variable.

A practical use of the low battery template code is to use it in an automation. I'll let you figure out how to do it. I implemented it to send a message to my phone each first day of the month at 9:00PM.

Note

When you create template variables in Home Assistant automations, the values are stored as strings, not as Python objects. So, to properly use template variables in Home Assistant automations you will need to use `tojson` and `fromjson` directives to serialize and deserialize the data properly.

We used `if/elif/else` and `for` to build complex template code including decision trees and actions that consider multiple states at once. Next, we'll put some emphasis on the Template Editor

and live debugging, so you can profile, test, and refactor templates quickly before using them into automations.

Using the Home Assistant Template Editor for debugging and testing

As you start writing more complex templates, testing them directly inside your configuration files can become difficult and time-consuming. To simplify this process, Home Assistant provides a built-in tool called the Template Editor, available under Developer Tools | Template. This tool allows you to safely write, test, and debug your templates in real time before integrating them into sensors, automations, or scripts.

When you open the Template Editor, you will see two main areas: the editor panel on the left, where you write your template code, and the result panel on the right, where the output is displayed instantly. As you type or modify your template, the result updates automatically, giving you immediate feedback. If there is a syntax error or an invalid expression, the result panel will show an error message, helping you quickly identify and fix the issue.

To understand how this works in practice, let's reuse the nested `if/elif/else` example introduced earlier in this chapter. Enter the template code into the editor panel, as shown in *Figure 3.2*. Once entered, the Template Editor evaluates the expression and displays the result on the right side of the screen.

Developer tools

YAML States Actions **Template** Events Statistics Assist

Templates are rendered using the Jinja2 template engine with some Home Assistant specific extensions.

- [Jinja2 template documentation](#)
- [Home Assistant template extensions](#)

Template editor

```

1 {% set t = states('sensor.kid1_room_temperature')|float(0) %}
2 {% if t >= 30 %}
3   It's very hot ({{ t }}°C). Turning on the fan to max.
4 {% elif t >= 25 %}
5   Warm ({{ t }}°C). Fan to medium.
6 {% elif t >= 21 %}
7   Comfortable ({{ t }}°C). Keeping current setting.
8 {% else %}
9   Cool ({{ t }}°C). Turning fan off.
10 {% endif %}

```

RESET TO DEMO TEMPLATE CLEAR

Result

Comfortable (23.5°C). Keeping current setting.

Result type: string

This template listens for the following state changed events:

- Entity: sensor.kid1_room_temperature

Figure 3.2 – Nested `if/elif/else` example in the Template Editor

In this example, the template reads the value of a temperature sensor and returns a message based on predefined conditions. The result panel not only shows the generated output but also provides useful context, such as the result type (for example, a string) and the entities involved which in this case is `sensor.kid1_room_temperature`. This visibility helps you verify that your template is correctly accessing and processing entity data.

By using the Template Editor, you can experiment with logic, validate syntax, and troubleshoot issues in isolation. This approach significantly reduces errors and makes it easier to build reliable templating code as your configurations grow in complexity.

Common template errors and fixes

You can use the Template Editor tool to detect error messages and debug your code by fixing the issues it reports. Different error messages can be presented according to the problem you have. I've compiled some of the messages I occasionally got while writing template code. These messages are presented in *Table 3.4*.

Error Message in Template Editor	Cause	Fix / Prevention
<pre>UndefinedError : 'None' has no attribute 'state'</pre>	Using <code>states.sensor.entity.state</code> when the entity is missing or unavailable	Use the safe function form: <code>states('sensor.kid1_room_temperature')</code>
<pre>TypeError: '>=' not supported between instances of 'str' and 'int'</pre>	Comparing string values to numbers	Convert to numeric using <code> float(0)</code> or <code> int(0)</code>

Error Message in Template Editor	Cause	Fix / Prevention
<pre>TemplateSyntax Error: expected token 'end of statement block', got 'elif'</pre>	Missing {% endif %} or incorrect Jinja block structure	Always close blocks properly: {% if %}...{% elif %}...{% else %}...{% endif %}
<pre>TemplateRuntim eError: No test named 'elifs'</pre>	Typo or extra "s" in Jinja keywords, or using a colon	Use correct syntax: {% elif condition %}
<pre>TypeError: must be real number, not str</pre>	Sensor returned 'unknown' or 'unavailable'	Handle safely with float(0) or int(0)
<pre>TemplateRuntim eError: 'float' object is not iterable</pre>	Attempting to use for over a numeric value	Only use for loops on lists or dictionaries, not single numbers
<pre>TemplateSyntax Error: Unexpected end of template. Jinja was looking for 'endif'.</pre>	Missing or mismatched control tag ({% if %} / {% endif %})	Verify each {% if %} has one matching {% endif %}

Error Message in Template Editor	Cause	Fix / Prevention
TemplateRuntimeError: division by zero	Division operation with a 0 denominator	Guard with an if-statement: <code>{% if value != 0 %}...{% endif %}</code>
TemplateRuntimeError: str object has no attribute 'state'	Treating an entity ID string as an entity object	Use <code>expand('entity_id')[0]</code> or access state via <code>states('entity_id')</code>

Table 3.4 – Template Editor error messages x causes x solution

Make sure to use the Template developer tool to validate your template messages in sensors and automations. They will save you time and help to check if you are getting the right information out of the template code.

We will see next a real application using templates that I implemented in my home to support a decision to invest in a new refrigerator for my house.

Real-world application: dynamic notifications, sensors, and UI customization

By the end of this section, you will have a complete system that monitors the energy consumption of a device, calculates its operational cost, triggers notifications when thresholds are exceeded, and visualizes this information in a Home Assistant dashboard. This solution combines template sensors, helpers, automations, and UI components to create a practical, data-driven workflow that supports real-time decision-making.

To demonstrate this approach, we will use power meter sensors not only to measure energy consumption but also to derive additional insights through templating. These derived values can be used in notification messages and dashboard cards, enabling you to identify when devices are consuming energy during peak hours. With this information, you can take action by controlling when devices operate, helping reduce overall energy usage and lower electricity costs.

In this example, I developed a complete infrastructure to monitor the power consumption of a refrigerator installed in my garage. It is an older appliance that came with the house, and the goal is to determine whether replacing it with a newer, more efficient model is financially justified. The block diagram representing this solution is shown in *Figure 3.3*.

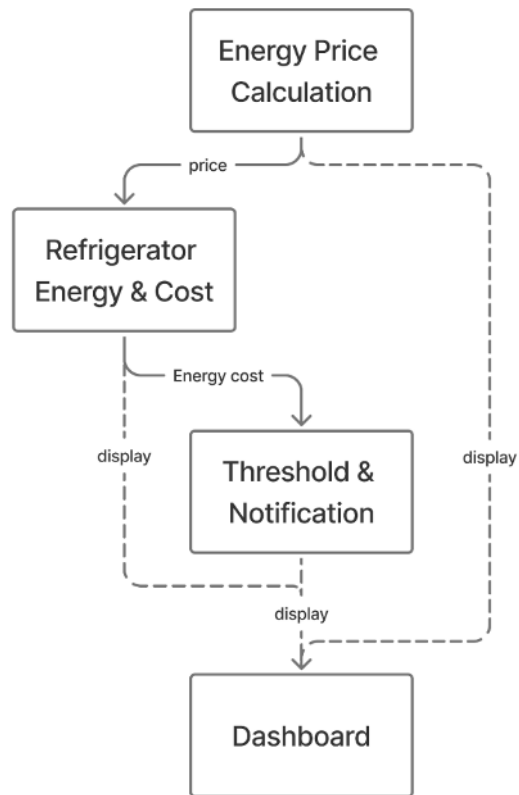


Figure 3.3 – Block diagram for the refrigerator garage energy monitoring solution

Let's explore each one of these blocks in the next subsections.

Energy price calculation

The breakdown block diagram for energy price calculation is represented in *Figure 3.4*.

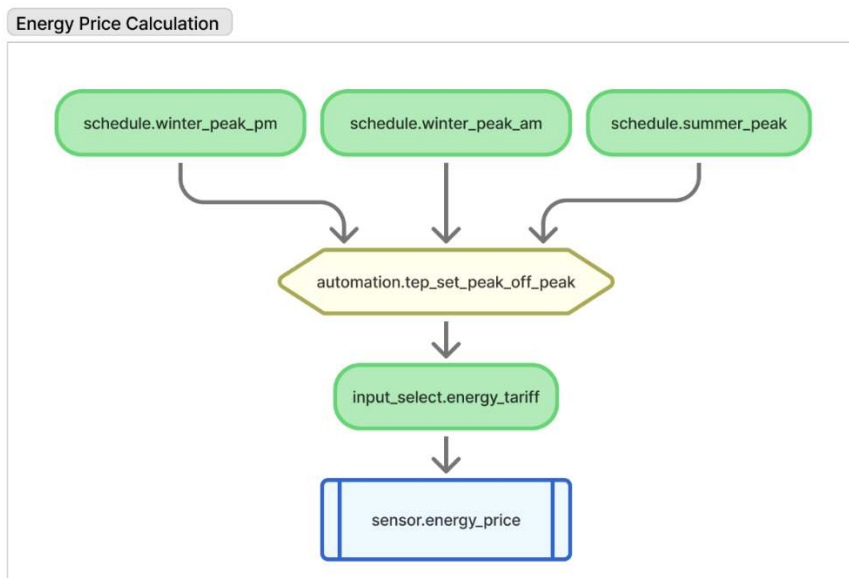


Figure 3.4 – Energy price calculation block diagram

The energy price calculation starts from the energy plan you have with your utility company. In my case the utility company will charge me differently during the peak time depending on the time for the day and depending if I'm in the summer or winter. In the winter, peak times are from 6-9AM and 6-9PM. In the summer, peak times are from 3-7PM. Both are during workdays. I had to create three schedule helpers (summer peak, winter morning peak, winter evening peak) to capture windows time during the day that it will be a peak or not. I used the weekday Integration that will test if the day is a workday or not.

I created an automation that watches for the schedule helpers, workday integration and months of the year to flip the tariff helper between "Peak" and Off-Peak". Using the energy tariff value, the template sensor energy price calculates the energy price for each time of the day. The code for the energy price sensor is shown in the sequence.

```
Template:
  sensor:
    - name: "Energy Price"
      unique_id: energy_price
      unit_of_measurement: "$/kWh"
      state: >
        {% set tariff = states('input_select.energy_tariff') %}
        {% set m = now().month %}
        {% set summer = m in [5,6,7,8,9] %} {# May-Sep #}
        {% if summer %}
          {# Summer rates #}
          {{ 0.1437 if tariff == 'Peak' else 0.0729 }}
        {% else %}
          {# Winter rates #}
          {{ 0.0843 if tariff == 'Peak' else 0.0758 }}
        {% endif %}
```

This block is generic which means the energy price sensor can be used as baseline to any other similar applications related to energy cost. What will going to change is just if you switch your energy plan to another type different than this. Of course, the tariff prices will have to be adjusted according to the values provided by the utility company you use.

The next subsection will describe the Refrigerator Garage energy and cost.

Refrigerator energy and cost

This block is represented in *Figure 3.5*.

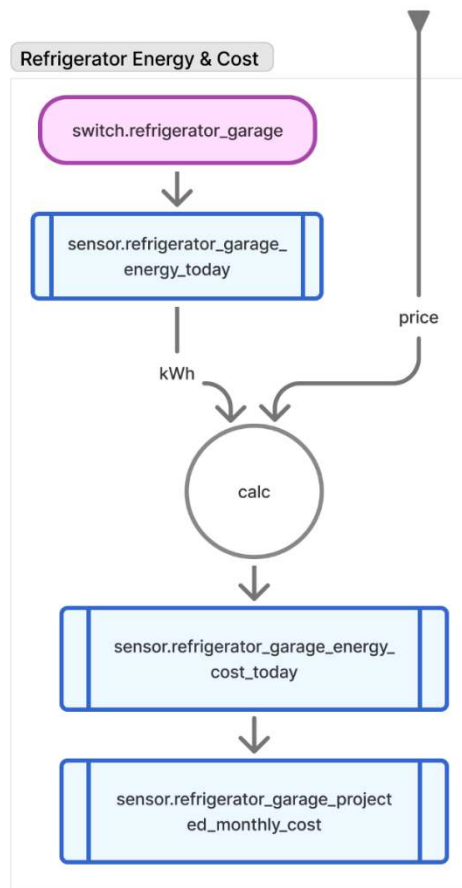


Figure 3.5 – Refrigerator garage energy and cost block diagram

My garage refrigerator is plugged into a power meter smart switch called `switch.refrigerator_garage`. It provides the entity `sensor.refrigerator_garage_energy_today` which gives me the amount of energy in kilowatt-hours (kWh) consumed during the day for the garage refrigerator. Using the price information from the `sensor.energy_price` calculated in the previous block, a simple calculation $\text{kWh} \times \text{price}$ will create another sensor called `sensor.refrigerator_garage_energy_cost_today`. This sensor gives me a cost in dollars that updates throughout the day according to the energy price. The YAML code for this sensor is shown below.

```
template
  sensor:
    - name: "Refrigerator Garage Energy Cost Today"
      unique_id: refrigerator_garage_energy_cost_today
      unit_of_measurement: "$"
      device_class: monetary
      state: >
        {% set kwh =
          states('sensor.refrigerator_garage_energy_cost_today')|float(0) %}
        {% set p = states('sensor.energy_price')|float(0) %}
        {{ (kwh * p) | round(2) }}
```

Another template sensor that I created projects the monthly energy cost of the device. This sensor is called `sensor.refrigerator_garage_projected_monthly_cost` and its template code is the following:

```
{% set spent = states('sensor.refrigerator_garage_energy_cost_today')|float(0) %}
{% set hour = now().hour + now().minute / 60 %}
{% set safe_hour = hour if hour > 0.1 else 0.1 %}
{% set daily_avg = spent / (safe_hour / 24) %}
{{ (daily_avg * 30) | round(0) }}
```

The next block will handle the threshold and notifications to send. We will explore them in the following subsection.

Threshold and notification

The block diagram that describes this subsection is presented in *Figure 3.6*.

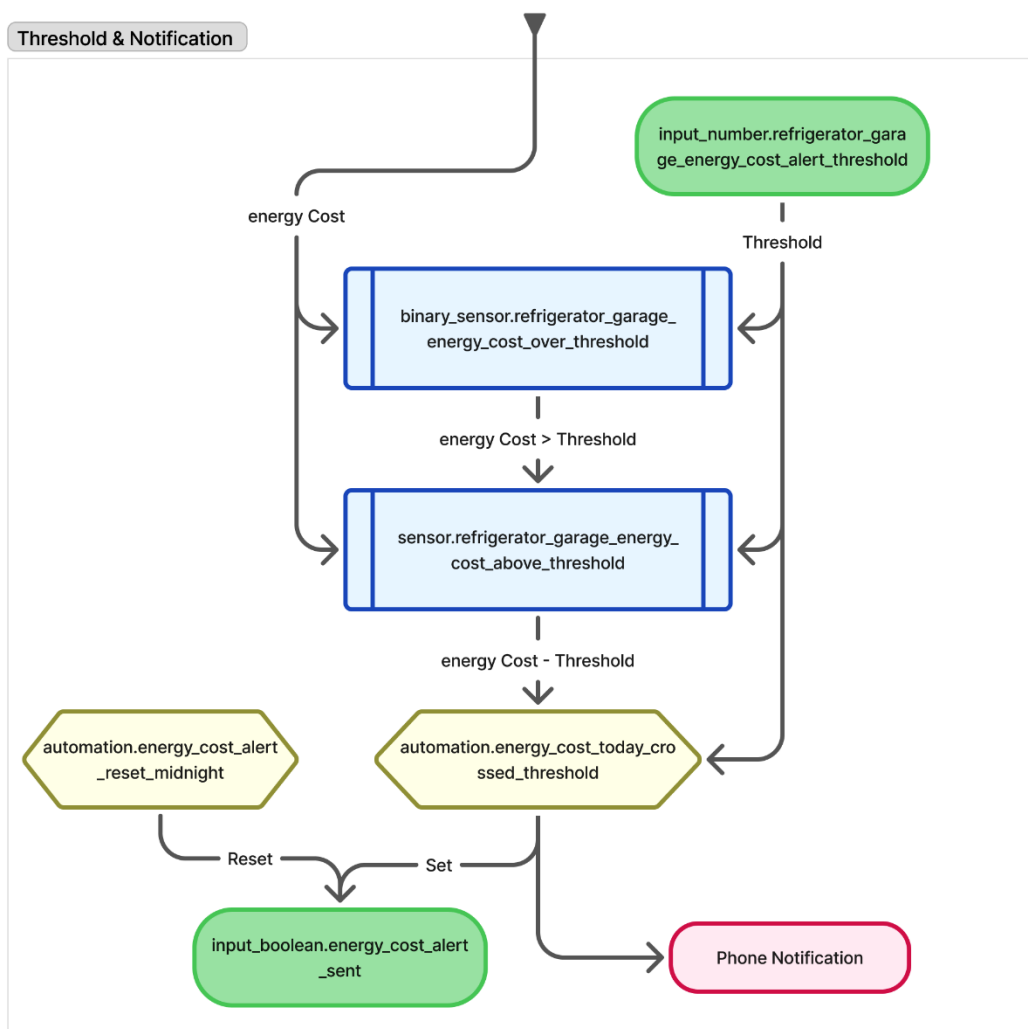


Figure 3.6 – Threshold and notification block diagram

The helper `input_number.refrigerator_garage_energy_cost_alert_threshold` sets the threshold limit for the Garage Refrigerator power consumption. This value is used as a parameter for two other sensors. One is a binary sensor (`binary_sensor.refrigerator_garage_energy_cost_over_threshold`) that turns on if the energy cost is above the threshold. The other sensor calculates how much above the limit the energy cost is. The `input_boolean.energy_cost_alert_sent` latch helper was created to avoid duplicated alerts, and it marks whether the notification has already been sent today. The

`automation.automation.energy_cost_today_crossed_threshold` sends a phone notification the first time the energy cost crosses the limit, and turns the cost alert sent latch on. The `automation.energy_cost_alert_reset_midnight` resets the latch just after midnight so a new alert can be sent the next day if it crosses the limit again.

The template code of the message sent to the phone in the `automation.energy_cost_today_crossed_threshold` is the following.

```
Cost today is $
{{    states('sensor.refrigerator_garage_energy_cost_above_threshold')|float(0)|
round(2) }}}, above your threshold of ${{
states('input_number.refrigerator_garage_energy_cost_alert_threshold')|float(0)|
round(2) }}.
```

This message can be tested using the developer tools discussed in the last section. In this case, testing the message gives me the following output:

```
Cost today is $0.07, above your threshold of $0.2.
```

The developer tools were used to test the outputs and validate the messages and content in other automations and template sensors developed in this section. In the next subsection, we will create a dashboard to present some data created in the previous subsections.

Dashboard

This subsection focuses on presenting the data generated in the previous steps, bringing together measurement, cost calculation, and threshold-based alerting into a single visual interface. At this stage, you already have the following components available: the calculated daily cost (`sensor.refrigerator_garage_energy_cost_today`), the projected monthly cost (`sensor.refrigerator_garage_projected_monthly_cost`), and the threshold-based alert logic derived from the helper (`input_number.refrigerator_garage_energy_cost_alert_threshold`).

To display this information, you can use the Markdown card in the Home Assistant UI, which supports templating dynamic content. This allows you to combine multiple data points and logic into a single card. *Figure 3.7* shows the diagram for this dashboard block.

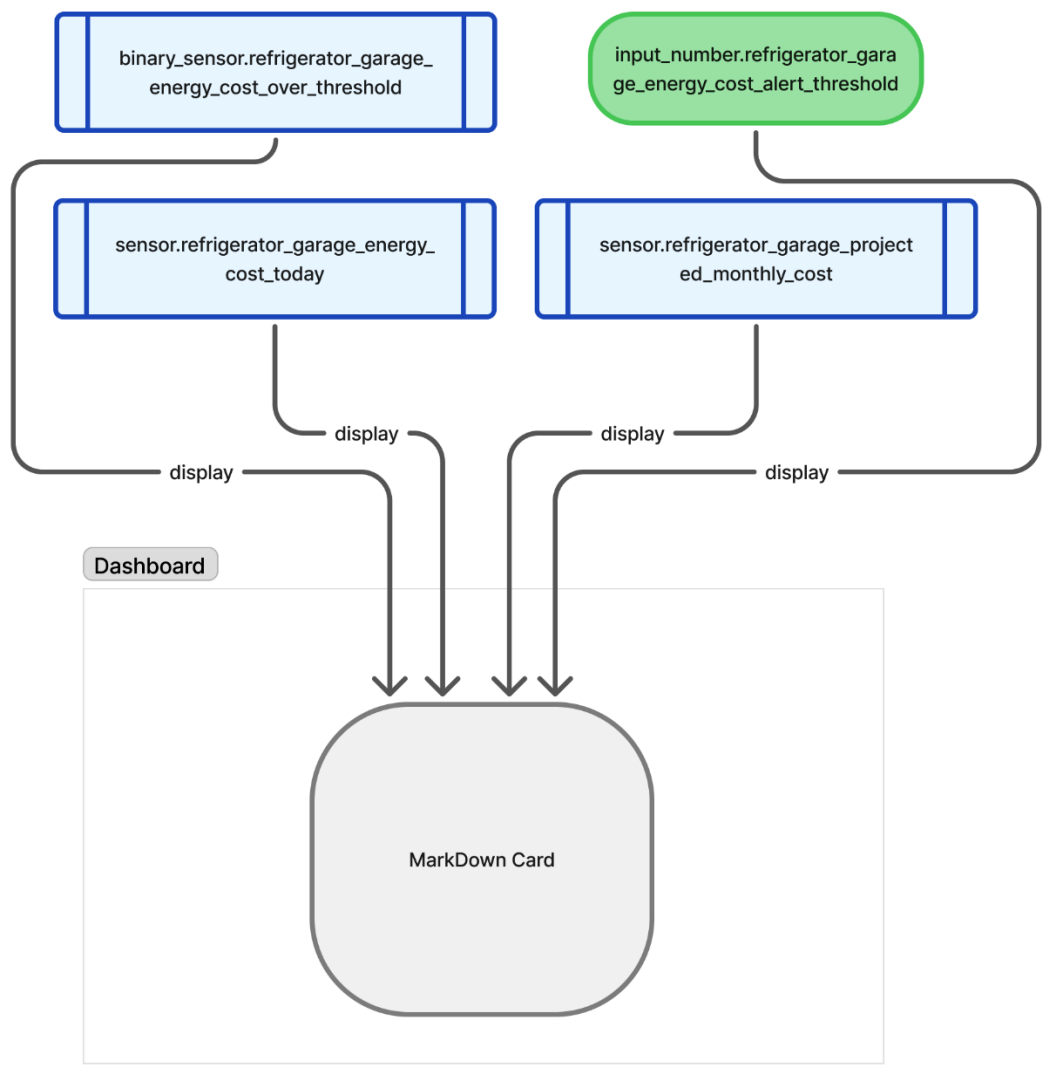


Figure 3.7 – Dashboard block diagram

Figure 3.8 presents the Markdown card configuration on the left and the corresponding rendered output on the right.

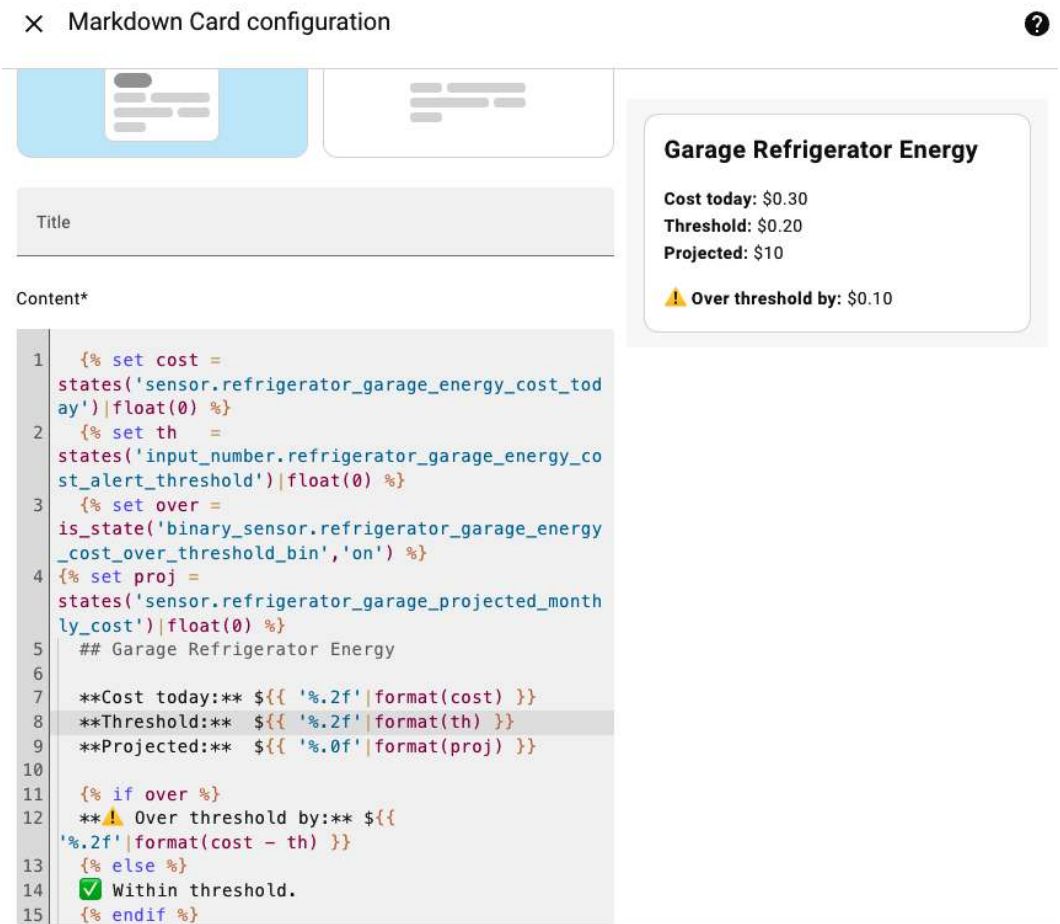


Figure 3.8 – Markdown card configuration

This dashboard consolidates the key elements created in the previous sections into a clear and actionable view. The card displays the real-time energy cost, the configured threshold, and the projected monthly estimate. It also evaluates the binary sensor `refrigerator_garage_energy_cost_over_threshold_bin` to determine whether the threshold has been exceeded and uses template logic to calculate how much the current cost surpasses that limit. This integrated view makes it easier to interpret the system's behavior and supports informed decision-making regarding energy consumption.

The overall solution follows a modular approach, where each component has a single responsibility: price calculation, energy measurement, threshold evaluation, and data presentation. This structure not only improves maintainability but also allows you to reuse the

same design for other devices by simply replacing the sensor names while keeping the pricing and alert logic unchanged.

Using the Developer Tools introduced earlier, you can validate and refine each template before integrating it into the dashboard, ensuring reliable templating code as your configuration grows in complexity.

We can now close this chapter, where we explored the powerful capabilities of Home Assistant templates. I hope you learned from and enjoyed the examples provided. This also closes the first part of the book, making room for the second part, which will focus on networking.

Summary

In this chapter, you learned how Jinja2 templating brings intelligence and flexibility to Home Assistant automations. Starting from basic syntax, variables, filters, and tests, the chapter showed how templates let your system respond dynamically to live data from entity states, attributes, and groups—transforming static configurations into data-driven automations.

You practiced using functions such as `expand()`, `selectattr()`, and `map()` to extract and process real-time data, and built template sensors that calculate averages, count open doors, or identify low-battery devices. The chapter then explored advanced templating with conditionals (`if`, `elif`, `else`) and loops (`for ... in`), teaching how to combine logic and data for more powerful automations. You also learned to use the Template Editor in Developer Tools to debug and fix template syntax or logic errors using real examples and a troubleshooting table.

Finally, a real-world project demonstrated how to use templating to monitor a refrigerator's power consumption, calculate energy costs, send dynamic notifications, and create a dashboard that visualizes data and alerts. This example tied together everything from template sensors and automations to UI customization.

In *Chapter 4, Securing Your Smart Home Network*, you'll explore smart home network security, learning how to protect your automation system through segmentation, encryption, and best practices for threat mitigation.

Part 2

Communication Protocols and Cloud Integration

In this part, you'll learn how to secure your smart home network, build and optimize a Zigbee mesh, and connect IoT devices to cloud platforms such as Golioth. By the end of this part, you'll be able to create a more secure, reliable, and connected home automation infrastructure.

This part of the book includes the following chapters:

- *Chapter 4, Securing Your Smart Home Network*
- *Chapter 5, Building Robust Zigbee Networks*
- *Chapter 6, Cloud IoT Integration with Golioth*

4

Securing Your Smart Home Network

Smart homes are becoming more powerful, but they are also becoming more exposed. Every connected device added to a home network, from sensors and cameras to smart plugs and voice assistants, increases the number of possible entry points for attackers. A poorly secured IoT device can expose private data, disrupt automations, or even provide a path into other systems on the same network. For advanced Home Assistant users, security is not only about protecting devices, but also about protecting privacy, reliability, and control over the entire smart home environment.

In this chapter, we will explore the most common vulnerabilities and threats found in IoT devices and home automation systems, including insecure defaults, weak credentials, outdated firmware, exposed services, and poorly isolated networks. We will then move into practical strategies for reducing risk, including network segmentation with Virtual Local Area Networks (VLANs) and firewall rules, secure remote access options, and device hardening techniques.

You will also compare secure remote access methods such as VPNs, Nabu Casa Cloud, and Cloudflare Tunnel, understanding where each approach fits and what trade-offs it introduces. The chapter will wrap up security monitoring, auditing, and incident response practices that can help you detect suspicious behavior and respond effectively when something goes wrong.

You will learn the following topics in this chapter:

- Understanding smart home vulnerabilities and threat landscape
- Network segmentation: VLANs and firewall rules for IoT isolation
- Secure remote access: VPN, Nabu Casa, and Cloudflare Tunnel

- Device hardening: firmware, passwords, and service management
- Security monitoring, auditing, and incident response

By the end of this chapter, you will be able to identify and mitigate common smart home security vulnerabilities, implement network segmentation using VLANs and firewall rules, configure secure remote access to Home Assistant, apply device hardening techniques, and develop a basic security monitoring and incident response plan for your Home Assistant setup.

Technical requirements

To get the most out of *Chapter 4*, you should have the following:

- Basic home networking knowledge, including what a LAN is, how devices receive IP addresses, and how a router connects and controls traffic between devices.
- Access to your router or firewall interface, so you can review your network settings and apply basic isolation or firewall rules when needed.
- Familiarity with Home Assistant configuration, including how to open and edit `configuration.yaml` and restart Home Assistant after changes.
- Basic command-line knowledge, enough to run simple commands such as ping to check whether devices can communicate with each other.
- Awareness of your Home Assistant installation type, such as Home Assistant OS, Docker Container, or a virtual machine installation.
- Basic understanding of MQTT, including the role of an MQTT broker and the idea that devices communicate by publishing and subscribing to topics.
- Basic understanding of Zigbee, including the role of a Zigbee coordinator, the concept of a mesh network, and whether your setup uses ZHA or Zigbee2MQTT.

You do not need to be an expert in networking, MQTT, or Zigbee. These requirements are intended only to help you follow the examples and understand how security decisions affect your Home Assistant environment.

Understanding smart home vulnerabilities and threat landscape

As the number of connected devices in our homes continues to grow, so does the attack surface available to potential intruders. While Home Assistant is designed primarily for local control and offline operation, modern smart homes increasingly depend on cloud services, APIs, and remote access. This section explores common vulnerabilities, typical attackers, and real-world breaches.

The chapter also introduces defense in depth, a security approach based on using multiple protection layers, such as device hardening, network segmentation, secure remote access, monitoring, and backups, to reduce the impact of a possible compromise.

Common vulnerabilities in IoT and smart home systems

A Home Assistant based smart home typically includes Wi-Fi, Zigbee or Z-Wave sensors, ESP custom devices, and various API integrations. Each of these layers can be exploited if not properly secured.

Even when Home Assistant itself is running locally, weak configurations in the surrounding network like open router ports or insecure IoT devices can expose the system to external threats. *Table 4.1* lists the common vulnerabilities in IoT and Smart Home Systems, including description and examples in the Home Assistant context.

Vulnerability	Description	Example / Home Assistant Context
Outdated Firmware	Many IoT devices stop receiving security updates after a year or two. Outdated firmware may contain known exploits.	Zigbee hubs or ESP devices running old firmware can be taken over remotely or crash when exposed to malformed packets.
Weak or Default Credentials	Default usernames like "admin" or "user" with weak passwords remain a top entry point for attackers.	Users running apps like Mosquitto, Samba, or MariaDB without changing default credentials risk immediate compromise.
Exposed Ports and Open APIs	Services left exposed on ports 8123, 1883 (MQTT), or 22 (SSH) are often scanned by bots within minutes of being online.	Exposing Home Assistant directly to the internet without a reverse proxy, VPN, or Nabu Casa is one of the most common misconfigurations.
Unencrypted Communication	Some devices transmit data in plain text, making it easy for attackers to intercept.	MQTT brokers without TLS or API calls without HTTPS expose credentials and states to sniffing.

Vulnerability	Description	Example / Home Assistant Context
Insecure Apps or Custom Integrations	Apps or third-party components can introduce vulnerabilities if not audited.	Malicious or outdated HACS integrations might leak data or execute unsafe commands.

Table 4.1 – Common vulnerabilities in IoT and smart home systems

Tip

Always review apps' GitHub repositories and avoid installing integrations from unknown sources. Stick to official or well-maintained HACS repositories.

Typical threat actors

In cybersecurity, threat actors are individuals or groups that intentionally exploit vulnerabilities to gain unauthorized access, disrupt systems, or steal information. In the context of smart homes, these actors range from opportunistic hackers using automated tools to scan the internet for exposed devices, to more sophisticated intruders targeting specific systems. Understanding who these attackers are and how they operate helps you, as a homeowner, anticipate risks and apply appropriate defenses, even when your systems are not directly exposed to the public internet.

Table 4.2 lists potential typical threat actors in the context of Home automation.

Threat Actor	Goal and Tactics	Example Scenario
Opportunistic Hackers	Use automated scripts to scan the internet for exposed devices or default passwords.	An unsecured Home Assistant port (8123) discovered by Shodan gets brute-forced using weak credentials.
Network Sniffers	Monitor Wi-Fi or LAN traffic to capture unencrypted data or credentials.	An attacker near your home captures MQTT traffic revealing device states and Wi-Fi details.
Malware Bots and Worms	Compromise poorly protected IoT devices to form botnets used for DDoS attacks or crypto mining.	An infected camera or router floods your network, slowing down or disconnecting your Home Assistant instance.

Threat Actor	Goal and Tactics	Example Scenario
Targeted Intruders	Rare in-home environments but may occur in high-value or business-linked setups.	A smart lock or camera used for monitoring office assets becomes the entry point for data exfiltration.

Table 4.2 – Typical threat actors

The data in *Table 4.2* highlights that most attacks on smart homes are not highly targeted, but rather opportunistic and automated. Tools like Shodan and large-scale botnets continuously scan the internet for exposed devices, weak passwords, or outdated firmware. Once discovered, these devices can be exploited for unauthorized access, data collection, or integration into botnets. While more advanced actors are less common in home environments, their methods such as network sniffing or targeted intrusion demonstrate why even local-only Home Assistant setups benefit from layered protection and proactive monitoring.

Case studies: real-world breaches

I did some research to bring you some cases that happened exposing Home Automation systems and devices. Here are some examples:

1. Ring Doorbell Account Takeover (2019): Attackers accessed home camera feeds due to reused credentials and lack of two-factor authentication.
2. Lesson: Enforce strong, unique passwords and enable **Multi-Factor Authentication (MFA)** wherever possible. Link for more information: <https://time.com/5748981/ring-security-cameras-hacked-safe-password/>
3. TP-Link Tapo L530E Smart Bulb Vulnerability (2023): Researchers performed a full vulnerability assessment and penetration test on the TP-Link Tapo L530E smart bulb, one of the best-selling Wi-Fi bulbs on Amazon. The study demonstrates that even simple smart home devices like bulbs can become powerful entry points for broader network compromise.
Lesson: Restrict IoT devices to isolated VLANs and block cross-network access. Link for more information here: <https://arxiv.org/pdf/2308.09019>
4. Home Assistant Community Incident (2021): A small subset of users reported exposed Home Assistant instances indexed by Shodan due to manual port forwarding.
5. Lesson: Avoid direct internet exposure; use Nabu Casa, VPNs, or secure tunnels like Cloudflare instead. Find more information here: <https://community.home-assistant.io/t/security-through-obscurity-isnt-a-good-strategy-on-the-internet/336791>

These real-world incidents highlight a consistent pattern: most smart home breaches occur not because of sophisticated explorations, but due to simple misconfigurations, weak authentication, and insecure device designs. Even a single exposed port or poorly secured IoT gadget can jeopardize an entire home automation environment.

As smart homes grow more interconnected, securing them requires more than isolated fixes. It demands a structured, layered strategy that anticipates failures and mitigates risk at every level. This leads us naturally into the next section, where we explore the Defense in Depth approach and how to apply it effectively within Home Assistant environments.

Defense in depth: a layered security approach

The principle of Defense in Depth is a well-established and widely recognized cybersecurity strategy. It originates from military defense concepts and has been formally adopted in information security by organizations such as NIST (National Institute of Standards and Technology) and ISO/IEC 27001.

In essence, Defense in Depth means protecting a system using multiple, overlapping layers of security controls, so that if one layer fails, others still provide protection. The idea is that no single defense is perfect but a combination of independent safeguards significantly reduces the likelihood and impact of an attack. The items in *Figure 4.1* give some example layers of how Defense in Depth applies to **smart homes and Home Assistant**:

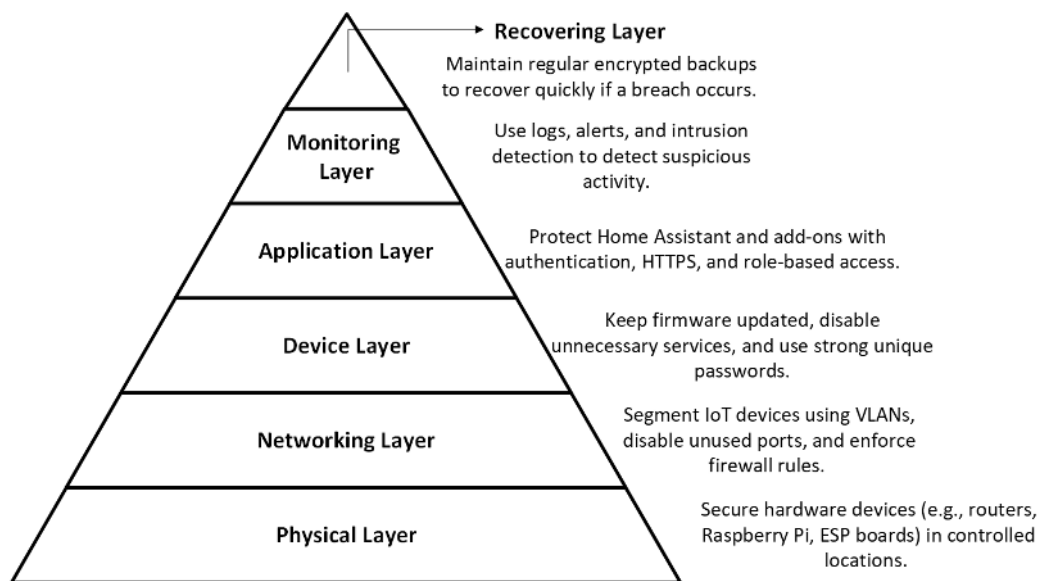


Figure 4.1 – Defense in depth example layers

These layers make clear that no single safeguard is enough to protect your smart home. Vulnerabilities in devices, misconfigured remote access, or weak network boundaries can each become a critical entry point. This is why Defense in Depth is essential for every Home Assistant deployment, regardless of whether it is exposed to the internet or kept fully local. One important layer of this approach is the network where its segmentation isolates IoT devices and limits the blast radius of any potential compromise. In the next section, we will explore how VLANs and firewall rules can be used to create a secure network foundation for your smart home.

Network segmentation: VLANs and firewall rules for IoT isolation

Most automated homes today contain dozens of IoT devices including lights, switches, plugs, cameras, vacuums, thermostats, and sensors. Many of these devices might run outdated firmware, expose lightweight web servers, or communicate insecurely. Without segmentation, a single compromised IoT device can become a pivot point for an attacker to access personal computers, NAS (Network Attached Storage), work laptops, or Home Assistant itself.

This section teaches you how to isolate IoT devices using VLANs, subnets, and firewall rules, applying the concept of defense-in-depth we saw in the last section.

VLAN and subnet fundamentals

A Virtual Local Area Network (VLAN) is a way to split a physical network into multiple isolated networks (subnets). Even though devices are connected to the same router, they behave as if they are on separate networks. A VLAN uses the following components:

- VLAN ID: Numeric identifier (1-4095) used inside the router
- Subnet: IP range assigned to each VLAN
- DHCP scope: provide automatic IP addresses for each VLAN
- Firewall rules: Control which VLANs can talk to each other and with internet.

As I have lots of IoT devices at home with different firmware which I don't know what they are doing besides being used as sensors or actuators, I decided to create a segmented network using VLANs at my home to isolate these IoT devices. I still allow IoT devices to reach the internet if they need to. I also can restrict the access of these devices to another VLAN where I can connect my personal computers, phones and my Home Assistant servers. I configured my Home Assistant servers to reach IoT devices when necessary.

Designing and implementing a segmented smart home network

In this section, we will use a practical example based on my own home network. The goal is to show how different device categories can be isolated while still allowing Home Assistant to communicate with the devices and services it needs.

The segmented network I created for my home is represented in *Figure 4.2*.

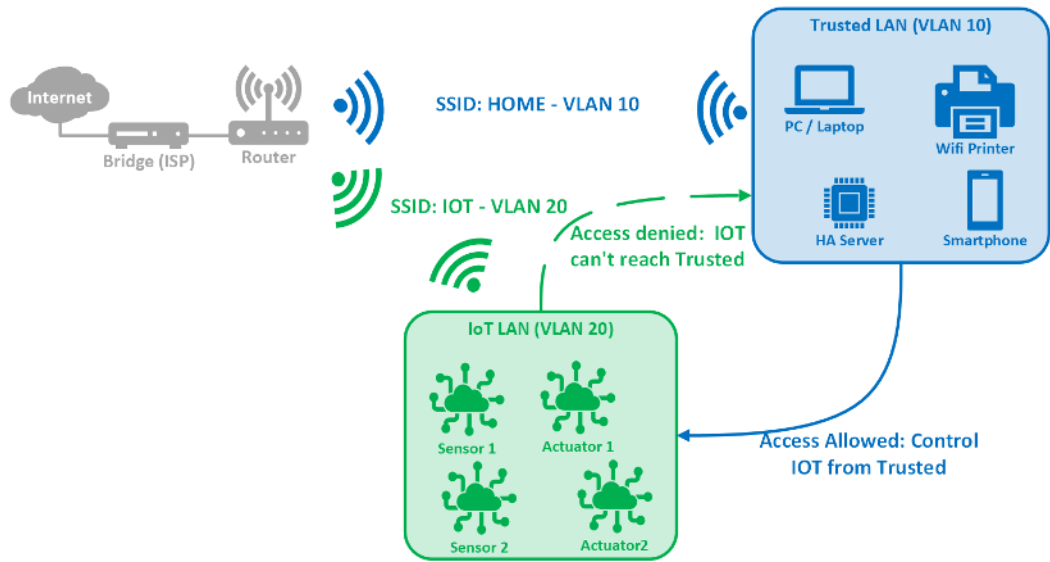


Figure 4.2 – Segmented network in my house

As can be seen in Figure 4.2, I created two VLANs which I will name as HOME and IoT. They have the following configuration presented in Table 4.3.

VLAN ID	Interface name	Description	Base IP
10	LAN (br-lan.10)	Main network	192.168.8.1
20	IoT (br-lan.20)	IoT devices network	192.168.20.1

Table 4.3 – VLANs configuration

We will break the VLAN creation and configuration into four subsections:

- Choosing your router
- Setting up the VLANs and creating the interfaces
- Configuring firewall and traffic routes
- Enabling IoT wireless interface

Choosing your router

I originally had an Arris SBG7400AC2 which is a combined modem/router that is supported by my Internet Service Provider (ISP). This integrated gateway doesn't have the features to create a segmented network which are: custom VLAN support, multiple LAN subnet support and firewall and traffic rules per VLAN.

The most favorable option for me when looking for a new router was the GL.iNet GL-MT6000 Flint2 model. This router offers all segmented network needs listed before and the following additional features:

- Wi-Fi 6 Technology
- Native OpenWrt operating system
- MediaTek Quad-core @2.0GHz CPU
- 1GB DDR4 RAM / 8GB eMMC Storage

The fact this router had all the required features and supports OpenWRT helped me to choose it. I had familiarity with OpenWRT before by hacking my router WRT54G model back in 2012. OpenWRT (<https://openwrt.org>) is an open-source headless Linux based operating system for embedded devices like wireless routers.

In my home network, the internet service provider router is an Arris cable modem/router. By default, this type of device can work as both a modem and a router, providing Wi-Fi, DHCP, firewall, and routing functions. However, for this project, I wanted the Flint 2 router to manage the smart home network because it supports more advanced configuration options, including VLANs and firewall rules.

To avoid having two routers managing the same network, I disabled the Wi-Fi and routing functions on the Arris device and configured it in bridge mode. In bridge mode, the Arris device works only as a modem. It passes the internet connection directly to the WAN port of the Flint 2 router. From that point forward, the Flint 2 becomes responsible for creating the local networks, assigning IP addresses, applying firewall rules, and managing the VLANs.

After you choose or verify that your router supports VLANs, you can move to the next topic to configure the VLANs and set up the network interfaces.

Setting up the VLANs and creating the interfaces

With the network design defined and the VLANs selected, it is time to create the logical interfaces that will represent each network segment within your router. In this subsection, we focus on the high-level concepts and workflow needed to build the segmented architecture shown earlier in *Figure 4.2*. The detailed step-by-step configuration of the Flint2 router will be demonstrated in *Video 4.1* (<https://packt.link/4024I>).

Before updating the router configurations or firmware, always create a backup of the current router configuration. On OpenWRT-based routers, firmware upgrades can usually preserve VLANs, firewall rules, DHCP settings, and network interfaces when the **Keep Settings** option is enabled. However, if the router becomes unstable after the update or configuration change, you may need to reset it and restore the configuration from backup. You can also export configuration files and store them in Git to track changes and document the network setup over time.

Note

On Flint2, the keep settings button is available on the router's configuration page, under SYSTEM | Upgrade | Firmware Online Upgrade.

To implement VLAN-based segmentation, you will generally follow these essential steps:

1. **Create the VLAN IDs:** Define one VLAN for your trusted network (e.g., VLAN 10) and another for the IoT network (e.g., VLAN 20). These identifiers will separate device groups at the network layer.
2. **Assign VLANs to a Bridge Interface:** add both VLANs to the LAN bridge used by your router. This step enables the router to handle tagged traffic while still allowing untagged devices to be mapped to the correct VLAN. *Figure 4.3* shows the VLANs configured in the Bridge Interface on the Flint2 router.

Bridge device: br-lan

General device optionsAdvanced device optionsBridge VLAN filtering

Enable VLAN filtering ☒

VLAN ID	Local	lan1	lan2	lan3	lan4	lan5	
		no link	no link	no link	no link	no link	
10	☒	u *	-	-	-	-	Delete
20	☒	-	-	-	-	-	Delete

Add

DismissSave

Figure 4.3 – VLAN configuration screen on Flint2 router

3. Create Logical Interfaces for Each VLAN like in *Figure 4.4*. Each VLAN requires its own interface with:
- A unique subnet (e.g., 192.168.8.0/24 for LAN, 192.168.20.0/24 for IoT)
 - DHCP configuration for automatic address assignment
 - A defined role in the firewall (LAN vs IoT zone): This interface-level separation allows the firewall to enforce segmentation later.
4. Configure DHCP and DNS Behavior: Ensure each VLAN interface provides DHCP services to its own devices. This separation prevents IoT devices from receiving IP addresses or DNS information intended for your trusted network.

Add new interface...

NameIOT

ProtocolStatic address

Devicebr-lan.20

CancelCreate interface

Figure 4.4 – Creating the IoT interface

At this stage, the logical foundation of the segmented network is complete. The router now recognizes and distinguishes the trusted and IoT networks, but communication policies must still be defined. In the next subsection we will create the rules that enforce isolation and allow only controlled communication between Home Assistant and your IoT devices.

Configuring firewall and traffic routes

Before we go over the firewall configuration, let's clarify some concepts related to the network traffic using OpenWrt and LuCi that will help you to understand better this section. The first concept is related to communication zones. We have three main interfaces and will define and setup three communication zones as following:

- WAN – internet boundary
- LAN – Local trusted network.
- IoT_ZONE – custom zone for our Home Automation devices.

The other concept we will have to explain is the settings rules applied to firewalls. These settings define the communication direction of the devices in each of these zones. *Table 4.4* presents, explains, and lists the questions addressed by these settings.

Setting	Explanation	Addressed Question
Input	Traffic coming from devices inside the same zone INTO the router itself	Can devices in this network communicate with the router?
Output	Traffic going from the router OUT to devices in the same zone	Can the router send traffic to devices in this network?
Forward	Traffic going through the router from one zone to another	Can devices in this network communicate to devices in other networks?

Table 4.4 – Firewall communication settings

Note

In this network design, IoT devices communicate through Home Assistant in VLAN10 instead of talking directly to each other. This keeps automations working while reducing lateral movement if one IoT device is compromised.

After applying these concepts, the end goal of this first part of the subsection is to have the table presented in *Figure 4.5* configured in Flint2 router.

Zones

Zone ⇒ Forwardings	Input	Output	Forward	Masquerading			
lan ⇒ wan	accept ▾	accept ▾	reject ▾	☐	≡	Edit	Delete
wan ⇒ REJECT	drop ▾	accept ▾	reject ▾	☒	≡	Edit	Delete
iot_zone ⇒ wan	reject ▾	accept ▾	reject ▾	☐	≡	Edit	Delete

Figure 4.5 – Firewall zones configuration

The recommended firewall configuration is to have the following policies:

- Default policy: IoT to LAN traffic is blocked. IoT devices should not initiate traffic toward the trusted network. This prevents compromised devices from attacking Home Assistant, NAS devices, laptops, and other sensitive systems.
- Allow LAN to IoT in certain cases. Your trusted devices like Home Assistant server often need to initiate connections toward IoT devices, for example, to send commands to smart plugs or bulbs.
- Allow IoT to access the Internet with restrictions. Many IoT vulnerabilities come from exposed services, weak default configurations, and devices communicating with unknown external servers. By allowing only outbound internet access, blocking unsolicited inbound traffic, and optionally forcing DNS requests through your router, you reduce the risk of remote attacks, unauthorized access, and malicious redirection while still keeping the devices functional.
- mDNS / SSDP Considerations: Many IoT ecosystems rely on discovery protocols like mDNS (Chromecast, HomeKit, ESPHome) and SSDP / UPNP (Smart TVs, streaming devices). Because multicast traffic does not cross VLANs by default, you may need an mDNS repeater like Avahi.

Figure 4.6 illustrates the traffic rules I had to create to accommodate the firewall restrictions.

Allow HA to IoT	Forwarded IPv4 and IPv6 From lan, IP 192.168.8.190, 192.168.8.179 To iot_zone	Accept forward	☒	 Edit Delete
Allow IoT to reach HA	Forwarded IPv4 and IPv6 From iot_zone To lan, IP 192.168.8.190, 192.168.8.179	Accept forward	☒	 Edit Delete
Allow mDNS IoT to Main	Forwarded IPv4 and IPv6, protocol UDP From iot_zone, port 5353 To lan, IP 224.0.0.251, port 5353	Accept forward	☒	 Edit Delete
Allow mDNS Main to IoT	Forwarded IPv4 and IPv6, protocol UDP From lan, port 5353 To iot_zone, IP 224.0.0.251, port 5353	Accept forward	☒	 Edit Delete
IOT Allow DNS	Incoming IPv4 and IPv6 From iot_zone To this device, port 53	Accept input	☒	 Edit Delete
IOT Allow DHCP	Incoming IPv4 and IPv6, protocol UDP From iot_zone To this device, port 67-68	Accept input	☒	 Edit Delete
IOT to LAN Forward reject	Forwarded IPv4 and IPv6 From iot_zone To lan	Reject forward	☒	 Edit Delete

Figure 4.6 – Traffic rules created

I will not go into detail on all these rules. You can follow *Figure 4.6* to implement them. I'd like to provide more details about the DNS and DHCP rules. They both should have Destination zone setup as Device (input) and Action as accept. DNS should have Destination Port as 53 and DHCP should have it as 67-68.

You will have to do an extra configuration for the mDNS protocol to work in the Flint2 router. You have to configure the avahi daemon by opening a terminal and issuing an SSH command to the router, like below:

```
ssh.root@192.168.8.1
```

Open and edit the Avahi configuration file `/etc/avahi/avahi-daemon.conf` using `vi` or `nano`. Look for a line `enable-reflector=no` and change it to `enable-reflector=yes`. Save the changes and exit the file editor. Restart the avahi service by issuing the command:

```
root@GL-MT6000:~# /etc/init.d/avahi-daemon restart
```

Check the Avahi service status using the command below. It should show as running:

```
root@GL-MT6000:~# /etc/init.d/avahi-daemon status
running
```

Note

In the Flint2 router, the Avahi daemon is installed by default. If you're using another OpenWRT-based router, make sure it is installed; if not, install it.

After doing the Firewall configurations you will need just one additional step to have your segmented network implemented in your home. You will have to enable the wireless interface for the IoT network to work. We will see next how to do it.

Enabling IoT wireless interface

Lastly, you will have to put your IoT wireless network interface to work. You will have to map each Wi-Fi SSID for the Lan and IoT to the proper VLAN interface. In the Flint2 router there is a workaround you have to execute to enable the IoT Wireless. This is described in the following sequence:

1. If a message "Wireless is not associated" is presented, you will have to take an extra step because the Wi-Fi SSID was created, but it never attached to a real radio interface. You will have to do it by opening a terminal and issuing the SSH command to the router as the following:

```
ssh.root@192.168.8.1
```

2. Using Vim or nano, edit the file `/etc/config/wireless` and add option `ifname 'ra2'` after the option key configuration in your wifi interface. You will have something like the following:

```
config wifi-iface 'wifinet4'
    option device 'mt798611'
    option mode 'ap'
    option ssid 'your_IoT_ssid'
    option encryption 'psk2'
    option key 'your_password'
    option ifname 'ra2'
```

```
option network 'IoT'  
option macaddr 'AB:CD:EF:12:34:56'
```

3. Save the file and return to the LuCi configuration screen. Reboot the router by going to **System --> Reboot**. You should now have your IoT network working and your Home Automation devices connected to it.

Because the Flint 2 router is based on OpenWRT and uses UCI for its network and firewall configuration, this workflow follows a common structure used by many OpenWRT-based routers. Other routers with VLAN, DHCP, wireless interface, and firewall-zone support follow similar concepts, although menu names and screen layouts may vary by manufacturer.

We finish here the creation of a segmented network. With this configuration, trusted devices are better protected from potential threats coming from less trusted IoT devices and networks.

When working with VLANs, interfaces, firewall zones, and wireless networks, small configuration mistakes can prevent traffic from flowing correctly. Common issues include assigning the wrong VLAN ID, using an incorrect IP range, connecting a device to the wrong SSID, forgetting to enable DHCP on the new interface, or blocking required traffic with firewall rules. A good troubleshooting approach is to test one layer at a time: first confirm that the device receives an IP address from the correct subnet, then test whether it can reach the router, then test internet access, and finally test whether Home Assistant can communicate with the device.

As a general guideline, configure and test one network segment at a time, document each VLAN ID and subnet, avoid overlapping IP ranges, and keep a backup of the router configuration before making major changes. This reduces human error and makes it easier to restore the system if a configuration change breaks connectivity.

Next, we will discuss how to secure remote access to your Home Assistant installation.

Secure remote access: VPN, Nabu Casa, and Cloudflare Tunnel

If you have your Home Assistant setup, you've probably tried accessing your smart home remotely, but exposing it to the internet using simple port forwarding is one of the most dangerous mistakes you can make. Attackers routinely scan the internet for open ports, and unsecured Home Assistant instances have been indexed by tools like Shodan in the past, leading to unauthorized access and compromised systems. This section explores secure alternatives for remote access, helping you maintain convenience without sacrificing safety.

Risks of direct port forwarding

Directly forwarding ports like 8123 from your router to Home Assistant creates several severe risks like:

- Exposure to Internet Scanners.
- No Protection Against Vulnerabilities
- Attacks on Apps or Other Services
- Increased Attack Surface, as port forwarding bypasses your firewall's normal isolation rules.

In practice, direct port forwarding exposes Home Assistant to unnecessary risk and should be avoided entirely. Instead, we will explore some secure methods in the next subsections.

VPN-based remote access (WireGuard or OpenVPN)

A VPN is one of the most secure methods for remote access because it avoids exposing Home Assistant to the internet entirely. Instead, you connect to your home network privately and access Home Assistant as if you were local.

There are two main popular VPN applications used: WireGuard and OpenVPN. WireGuard is an efficient, modern, and lightweight VPN protocol. It is fast because it uses modern cryptography. OpenVPN is older but more flexible. It is an open-source VPN solution that provides secure, encrypted remote access to your Home Assistant network.

The Flint2 router supports both WireGuard and OpenVPN in server and client mode. These two modes are explained as follows.

VPN - server mode

In server mode, the router itself becomes the VPN server. You, or anyone you authorize, can connect to your home network remotely using a WireGuard or OpenVPN client on a phone, laptop, or tablet. This approach keeps Home Assistant hidden from the public internet because you first connect securely to your private home network, and then access Home Assistant as if you were at home.

Using a VPN in server mode provides strong security because it encrypts the connection between your remote device and your home network. However, it requires installing a VPN client on each remote device and may be less convenient for non-technical household members.

I created *Video 4.2* (<https://packt.link/e2DZv>), where I explain how to configure the Flint 2 router to work with WireGuard in server mode.

VPN – client mode

In client mode, the router behaves like a device connecting to a commercial VPN service such as Mullvad, ProtonVPN, Surfshark, or NordVPN. The router establishes an encrypted outbound connection to a remote VPN server, and selected devices in your home can route their internet traffic through that tunnel.

Client mode is useful when you want to protect outbound traffic from devices inside your home, but it is not the typical method for remotely accessing Home Assistant from outside your network.

Figure 4.7 shows both VPN modes configured in the router. The connection at the top is configured in server mode, and the connection at the bottom is configured in client mode.

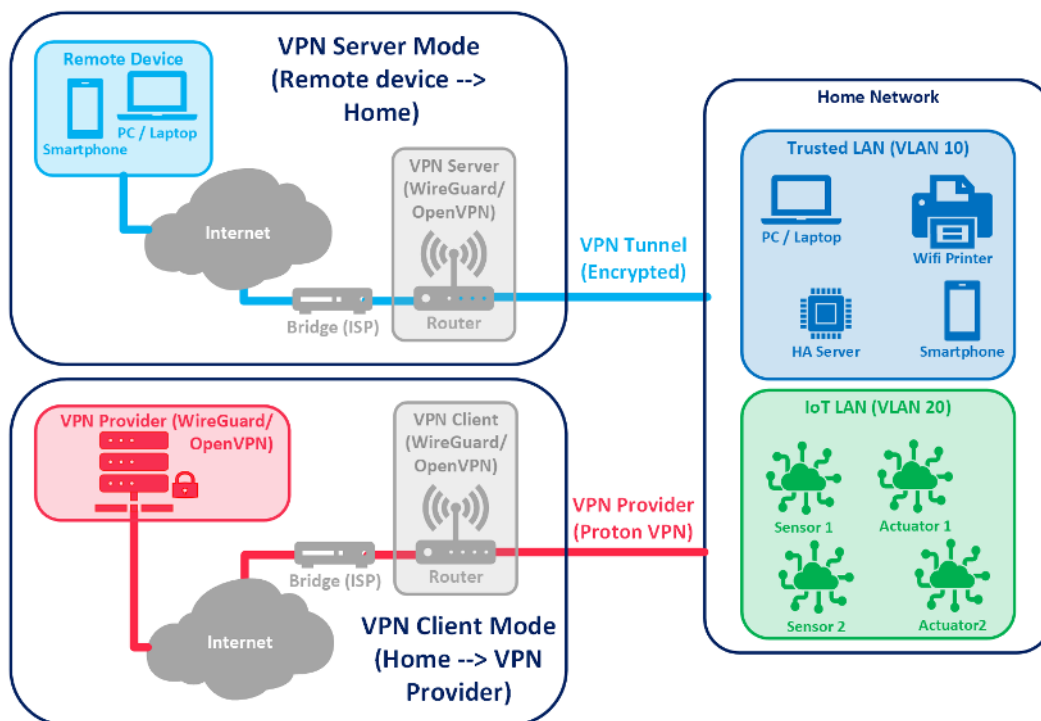


Figure 4.7 – VPN connection modes

If you prefer simple configuration steps and full support within Home Assistant, you should choose Nabu Casa which we will present in the next section.

Nabu Casa remote access

Nabu Casa (<https://www.nabucasa.com>) is the official paid cloud service for Home Assistant, offering secure remote access without requiring users to configure routers, SSL certificates, or tunnels. It provides a simple and reliable way to reach your Home Assistant instance from anywhere.

One of Nabu Casa's main advantages is how easy it is to use. Nabu Casa requires no port forwarding and delivers fully encrypted remote access. It works seamlessly with Google Assistant and Alexa, supports secure API communication for mobile apps, and includes additional Home Assistant Cloud features such as cloud hooks and voice services.

Nabu Casa is best suited for users who want secure, hassle-free remote access and are comfortable with a low-cost subscription in exchange for convenience and simplicity.

Cloudflare Tunnel: free, secure, and powerful

Cloudflare Tunnel offers free and secure remote access to Home Assistant without exposing your IP address or requiring any open ports. It works by running a lightweight cloudflared agent (<https://github.com/brenner-tobias/addon-cloudflared>) on your Home Assistant host, which creates an outbound, encrypted connection to Cloudflare. Cloudflare then provides a public HTTPS URL that is fully protected by its security infrastructure.

Cloudflare Tunnel also has solid security advantages. Because it proxies all inbound traffic, attackers never directly reach your home network. Even if someone obtains your public URL, you can apply Zero Trust policies including Google or Microsoft single sign-on, hardware key authentication, and IP filtering to ensure only authorized users gain access. However, one practical disadvantage is that you must own and manage a domain name to use Cloudflare Tunnel effectively, which adds an extra step and cost for some users.

Managing SSL certificates with Let's Encrypt and DuckDNS

If you prefer to terminate HTTPS locally, Let's Encrypt offers free SSL certificates that you can use with Home Assistant. One common approach is to pair Let's Encrypt with DuckDNS, which provides a free dynamic DNS hostname that updates automatically when your public IP changes. The DuckDNS app can automatically request and install Let's Encrypt certificates and handle renewal for you, making it easy to maintain secure HTTPS access.

However, this method typically requires port forwarding to complete certificate validation, unless you combine it with a reverse tunnel like Cloudflare, which supports DNS-based challenges and avoids opening ports. Because of this added complexity, DuckDNS and Let's Encrypt are recommended only for users who are comfortable managing certificates and configuring firewall rules manually.

Table 4.5 summarizes the secure remote access methods to be used in your Home Automation network.

Method	Difficulty Level	Port Forwarding	Notes
VPN	Medium	No	Full LAN access
Nabu Casa	Easy	No	Recommended for most users
Cloudflare Tunnel	Medium	No	Free, requires domain
DuckDNS + Let's Encrypt	Medium	Yes	Need Cert renewal

Table 4.5 – Remote network access methods

All of these approaches to improving the security of your home network aren't enough if a device inside it has a security breach, for example. We will see how to handle this type of situation and others in the next section.

Device hardening: firmware, passwords, and service management

Even with strategies to increase your network security discussed in the previous section, individual devices can still become the weakest link in a smart home. IoT breaches often originate from the simplest devices like sensors and plugs especially when they use outdated firmware or weak credentials. Device hardening focuses on reducing attack surfaces at the device level, ensuring each component in your system is resilient even if another layer fails. It is part of the device layer under the Defense in Depth approach we discussed in the first section of this chapter.

In this section, we will discuss practical techniques for strengthening safety for IoT devices.

The importance of firmware updates

Firmware is the IoT device's operating system, and vulnerabilities in it are one of the most common components for attacks. Many IoT vendors have poor update policies, meaning a device may remain vulnerable for years unless manually checked.

Firmware updates can patch vulnerabilities, fix bugs affecting stability or battery use, improve Wi-Fi, Zigbee, or BLE connectivity and add security controls such as encrypted OTA or MQTT. So, in the end, keeping firmware updated comes with many improvements to your IoT device.

I recommend you to always update devices before onboarding them into Home Assistant. Also prefer open-source or well-maintained firmware (ESPHome, Tasmota, Zigbee2MQTT-supported devices). You can even track firmware update cycles using a Home Assistant reminder automation.

Dealing with passwords

It goes without saying that default passwords remain the leading cause of IoT device compromise. As best practice, try to change all default admin credentials during onboarding, before adding the device to Home Assistant. Also use unique passwords per device, managed through a password manager, and rotate passwords annually or after any suspected network breach. As we discussed in *Chapter 2*, avoid storing passwords directly in YAML and use Home Assistant `secrets.yaml` instead.

As general guidelines for passwords:

- Keep them at least 16 characters
- Combine upper/lowercase, numbers, symbols
- Never reuse Wi-Fi passwords as device passwords

Disabling unused services (Telnet, SSH, Web UI)

Every exposed service increases your attack surface. Many devices ship with services enabled by default, including Telnet, unsecured web interfaces, SSH, or over-the-air (OTA) update ports. If these services are not protected or no longer needed during normal operation, they can become entry points for unauthorized access.

Some recommended actions are:

- Disable Telnet or any other legacy protocol when possible.
- Disable the device's web UI if it is no longer needed after provisioning.
- Restrict SSH access to trusted admin devices or disable it if it is not required.
- Prefer encrypted protocols, such as HTTPS and MQTT over TLS, when supported.
- Treat OTA update services carefully. Do not disable OTA unless you have another reliable firmware update method, such as USB or serial flashing. OTA security is covered in more detail in the next section.

In Tasmota devices, the following commands, issued from the console, control access to some services:

- `SetOption5 1`: Disable web server password reset
- `Webserver 2`: restrict Web UI to admin only
- `SetOption3 1`: Disable legacy MQTT

In ESPHome devices, including port: 0 to the web_server: configuration fully disables the web server.

Secure OTA updates and device renaming

OTA updates are convenient because they allow firmware updates without physically connecting to the device. However, they can also create security risks if they are not protected. If an attacker gains access to an unsecured OTA mechanism, they may be able to install unauthorized or malicious firmware.

Some recommended secure OTA practices are:

- Enable OTA passwords or encryption when supported.
- Avoid plain HTTP OTA update servers.
- Do not expose OTA ports outside the IoT VLAN.
- Perform OTA updates only from trusted devices or trusted network segments.
- Monitor or log OTA-related activity when possible.

Device renaming also improves security and manageability. Default names can reveal vendor or model information, making it easier for an attacker to identify known weaknesses. Clear custom names also help you identify devices quickly during troubleshooting or incident investigations.

For more guidance about device names, refer to the best practices for folder structure and naming conventions discussed in *Chapter 2*.

Monitoring new devices connecting to the network

Hardening devices is only half the job. You must also detect when unknown devices attempt to join your network. Some techniques you can use to do that includes:

- Enable alerts for new DHCP leases on your router.
- Use Home Assistant integrations to detect new devices like UniFi Network, Fritz!Box Tools or ASUSWRT
- Set automations for unknown MAC addresses to Notify your phone or Block the device automatically if supported. An example using the Unifi Controller Integration can find here: <https://community.home-assistant.io/t/get-notified-of-new-devices-on-your-network/>

Device hardening enhances each individual component of your smart home, ensuring that sensors, switches, and custom devices remain secure even if other layers of protection are challenged. By keeping firmware updated, enforcing strong credentials, disabling unnecessary

services, and securing OTA workflows, you significantly reduce the likelihood of a device becoming an entry point for attackers.

To support these practices, a Device Hardening Checklist is included at the end of this chapter, giving you a practical reference you can use for every new device you deploy.

In the next section, we move from strengthening devices to actively detecting and responding to threats, exploring Security monitoring, auditing, and incident response to help you recognize unusual activity and react quickly when something goes wrong.

Security monitoring, auditing, and incident response

Even a well-secured smart home can run into surprises from time to time. A device might start acting different, an integration may stop behaving as expected, or a new device could show up on your network without explanation. That is why security monitoring is an ongoing part of keeping your system healthy. In this section, we'll look at how Home Assistant and its apps can help you keep an eye on what's happening, spot unusual behavior, and handle potential security issues with a clear and repeatable process.

Using Home Assistant and Apps for log and event monitoring

Home Assistant provides several mechanisms to track the health and security of your installation. The main ones that I used are listed below:

- **Home Assistant Logs:** The core logging system captures warnings, authentication failures, integration issues, and connection errors. Go to **Settings | System | Logs** to access it. Use the search tool to isolate failures or security-related entries.
- **Supervisor & App Logs:** When using Home Assistant Operating System (HAOS) or supervised installs, the Supervisor and apps (e.g., Mosquitto, Zigbee2MQTT) maintain their own logs. These can reveal, for example MQTT clients connecting unexpectedly, Zigbee devices re-pairing without your action or excessive restart loops indicating potential tampering or instability.
- **Events Monitoring Using the Developer Tools:** **The Developer Tools | Events tab** allows you to listen for system-wide events such as `home_assistant_started`, `device_tracker.new_device`, `state_changed`, `automation_triggered`. This is extremely helpful during investigation review or when building detection automations. We will do this later in this section.

Detecting suspicious behavior

We can monitor security in Home Assistant by noticing deviations from regular operation. Some examples of these are:

- **Unknown Devices Joining the Network:** New MAC addresses joining Wi-Fi, New Zigbee devices attempting to pair, New MQTT clients connecting to the broker. These events can be detected by tools like UniFi, OPNSense, Zigbee2MQTT logs, or the Mosquitto broker logs.
- **Failed Login Attempts:** Repeated login failures, especially from external IPs, can indicate attempted brute-force access. To detect these enable advance mode in Home Assistant to see the more in deep logs and review entries like Login attempt failed and Login attempt with IP address X.X.X.X.
- **Traffic Spikes or Abnormal Message Rates:** Unusual network behavior is often overlooked in home environments. A compromised or malfunctioning IoT device may publish MQTT messages at extremely high frequency, flood your router with traffic or cause latency in automations. Some tools you can use to detect these are MQTT Explorer (which we will see in *Chapter 8*), the Glances app (seen in *Chapter 1*) to detect CPU/network spikes, or Grafana dashboards monitoring network or database behavior.

Building alerts with automations and notifications

Home Assistant can act as a security sentinel through simple but powerful automations. These alerts can be sent to: Mobile apps using Home Assistant Companion, Telegram, E-mail, Persistent notifications, Slack or Discord.

Examples of alerts that can be implemented are:

1. New device detected on the network
2. Multiple failed login attempts
3. MQTT unknown client connected
4. Zigbee device enters pairing mode unexpectedly
5. Critical integration stops responding
6. Apps crashes repeatedly

We will implement the automation mentioned in item 1 above. We have to go to **Settings | Automation & Scenes | + Create Automation**. Include the YAML code in the sequence and click on the Save button.

```
alias:
  Alert on New Network Device
```

```

description:
  Sends a notification when a new device joins the network.
triggers:
  event_type: device_tracker.new_device
  trigger: event
conditions: []
actions:
  action: notify.mobile_app_iphone
  metadata: {}
  data:
    title: "Security Alert: Unknown Device"
    message: >-
      A new device has joined your network.
      Name: {{ trigger.event.data.host_name }}
      MAC: {{ trigger.event.data.mac }}
      Source: {{ trigger.event.data.entity_id }}
  mode: single

```

You can test your automation by adding a new device to your Home Assistant installation or going to **Developer tools** | **events** and fill the following fields:

- Event Type: `device_tracker.new_device`
- Event data: `host_name: test`
`mac: 00:00:00:00:00:00`
`Entity_id: HA`

Click on the **Fire Event** button. Your event will be sent, and if everything is correct with your automation, you should receive a message on your phone.

Creating a simple incident response workflow

Incident response originally comes from professional IT and cybersecurity environments. In those settings, it refers to a structured way of dealing with security issues: detection, containment, investigation, and recovery. Large companies use it for data breaches, malware infections, unauthorized access, and other high-impact incidents. The core idea of incident response is much simpler:

Note

Something unexpected happened: you want a repeatable way to deal with it.

In home automation, while we don't use the same heavy frameworks as enterprise IT, the routine is essentially the same but scaled down. *Table 4.6* lists a five-stage action protocol to deal with network incidents in Home Automation:

Stage	Description	Tools / Actions
Detect	Identify abnormal activity or alerts	Logs, automations, Grafana, Uptime Kuma
Contain	Stop the issue from spreading	Disable integrations, block MAC/IP, isolate VLAN
Eradicate	Remove the root cause	Reflash firmware, delete rogue entities
Recover	Return system to normal operation	Restore backups, re-enable integrations
Lessons Learned	Improve future security	Add alerts, rotate credentials, adjust firewall

Table 4.6 – Incident response protocol to deal with home automation network issues

Even in a well-secured smart home, unexpected behavior can still occur, and having a simple incident response routine helps you deal with those situations quickly and confidently. *Table 4.6* summarizes and consolidates all the practices discussed in the chapter, bringing them together into a clear and practical workflow you can apply whenever something goes wrong in your Home network.

Summary

In this chapter, you learned how to increase the security of a Home Assistant environment by identifying common vulnerabilities in IoT ecosystems and understanding how real-world security incidents can affect smart home devices.

You also explored the concept of defense in depth and learned how multiple protection layers, from physical security and network segmentation to monitoring and backups, work together to reduce risk. You practiced designing a segmented smart home network using VLANs and firewall rules, with a practical example based on an OpenWRT-powered Flint 2 router.

You compared secure remote access options, including VPNs, Nabu Casa Cloud, and Cloudflare Tunnel, and evaluated each approach based on security, complexity, and network exposure.

You also learned device hardening techniques such as updating firmware, using strong passwords, disabling unnecessary services, and reducing attack surfaces.

Finally, you learned how to monitor security events, analyze suspicious behavior, create alerting automations, and build a lightweight incident response process for a home environment. The chapter concluded with a security checklist that you can use as a quick reference when deploying or reviewing smart home devices.

In *Chapter 5, Building Robust Zigbee Networks*, we will explore how Zigbee networks operate, including coordinators, routers, end devices, channels, and mesh routing. You will also learn how to configure and optimize Zigbee2MQTT and ZHA in Home Assistant, troubleshoot common connectivity issues, and build a practical Zigbee-based automation project.

Checklist – device hardening quick steps

Before Adding Device to Home Assistant

- Update firmware to latest version
- Change all default credentials
- Rename device using secure naming scheme
- Disable unneeded services (Telnet, web UI, mDNS)

During Onboarding

- Set strong API/OTA passwords
- Enforce encrypted communication (ESPHome API, MQTT TLS)
- Assign device to IoT VLAN
- Document device purpose and links

Post-Onboarding

- Set Home Assistant alerts for new devices
- Review logs monthly
- Rotate passwords annually
- Verify OTA remains protected

5

Building Robust Zigbee Networks

A reliable smart home depends not only on good automations, but also on a stable communication network between devices. While Wi-Fi is common, it is not always the best option for battery-powered sensors, buttons, switches, and distributed devices that require low power consumption and dependable coverage. Zigbee solves many of these challenges by using a mesh network, where devices can route messages and extend the reach of the system.

In this chapter, we will explore how to build, optimize, and maintain a scalable Zigbee mesh inside Home Assistant. We begin by explaining the Zigbee protocol, including channels, mesh routing behavior, and the roles of coordinators, routers, and end devices.

You will then move into the practical implementation of Zigbee using two widely adopted Home Assistant integrations: Zigbee2MQTT (Z2M) and Zigbee Home Automation (ZHA). You will learn how to configure these integrations, evaluate network performance, and apply optimization techniques for long-term reliability.

The chapter concludes with a hands-on project that combines a Zigbee button with an ESP8266-based Wi-Fi relay board running Tasmota to control a garage gate.

By the end of this chapter, you will be able to understand the fundamentals of Zigbee mesh networking, configure and optimize Zigbee2MQTT and ZHA integrations in Home Assistant, design and build a robust and scalable Zigbee network, integrate Zigbee and Wi-Fi devices into a unified automation solution, and wire an ESP8266 to control a garage door motor safely using Tasmota firmware.

We will cover the following topics in this chapter:

- Zigbee protocol fundamentals: device types, mesh, and channels

- ZHA (Zigbee Home Automation): setup and optimization
- Zigbee2MQTT: advanced configuration and device management
- Building a robust mesh: router placement and network health
- Hands-on project: integrating Wi-Fi and Zigbee for garage gate control

Technical requirements

To follow this chapter effectively, you will need the following:

- A working Home Assistant installation with either ZHA or Zigbee2MQTT available
- A Zigbee coordinator, preferably a USB dongle connected to your Home Assistant system
- At least one Zigbee router-capable device to help build and strengthen the mesh network
- Basic familiarity with Home Assistant device management, YAML configuration, and apps
- For the hands-on project, an ESP8266 or ESP32 board with a relay
- A garage door opener that supports dry-contact input control
- Basic wiring skills to connect the relay safely to the garage door opener
- Familiarity with installing, configuring, and using Tasmota on the ESP8266 or ESP32 board

Zigbee protocol fundamentals: device types, mesh, and channels

Zigbee is a wireless communication protocol built on the IEEE 802.15.4 standard, which was first ratified in 2003. Its strengths are reliability, mesh coverage, and energy efficiency rather than high throughput. These qualities make it ideal for battery-powered IoT devices and dense smart home environments. As a result, Zigbee has become one of the preferred protocols for home automation.

Table 5.1 compares the main wireless protocols used in home automation.

Protocol	Main Advantage	Limitations	Typical Applications
Zigbee	Low power consumption, mesh networking, large device ecosystem	Shares the 2.4 GHz band with Wi-Fi	Sensors, switches, smart plugs, lighting
Wi-Fi	High bandwidth and direct IP connectivity	Higher power consumption and no native mesh support for IoT devices	Cameras, speakers, ESPHome devices, high-data applications
Bluetooth/BLE	Very low power and simple device pairing	Short range and limited scalability	Wearables, beacons, proximity sensors
Z-Wave	Operates on sub-GHz frequencies, reducing Wi-Fi interference	Smaller device ecosystem and generally higher device cost	Reliable home automation networks with minimal RF interference

Table 5.1 – Comparison of wireless protocols used in home automation

As shown in *Table 5.1*, each wireless technology has strengths that make it suitable for different applications. Zigbee offers a good balance of power efficiency, mesh networking, device availability, and Home Assistant support, making it one of the most widely used protocols for home automation.

Thread is an emerging IPv6-based mesh protocol built on the same 802.15.4 layer as Zigbee and forms the foundation for Matter. While Thread adoption is growing, Zigbee remains extremely popular due to its affordability, extensive device availability, and mature integration with Home Assistant. For most smart homes today, Zigbee continues to be a reliable and cost-effective choice. In the next subsections, we will discuss Zigbee network components, how they are interconnected, and the communication channels used by it.

Device Types

The Zigbee network is composed of three main device types: coordinator, router, and end device. These devices work together to create and maintain the mesh network. The coordinator starts and manages the network, routers extend coverage by forwarding messages between

devices, and end devices communicate through the mesh while usually conserving power. Next, we will look at each device type in more detail.

Zigbee coordinator

The coordinator is the central component of a Zigbee network. There is just one coordinator in the Zigbee network, and its function is to create and manage the Zigbee PAN (Personal Area Network). You will easily find Zigbee coordinators as USB dongles that you can plug directly into your Home Assistant server, which is a practical way to start your Zigbee network. There are also network-based coordinators that use an Ethernet or Wi-Fi interface. They require an additional network dependency but are flexible in terms of physical placement which can allow a central location for better mesh symmetry.

The coordinator devices I use are the Sonoff ZBDongle-E based on the chipset Silicon Labs EFR32MG21 but there is a new device from this same brand called MG24 based on the Silicon Labs EFR32MG24 that is also recommended and supported by Home Assistant. A list of recommended radio adapters and modules to be used with Home Assistant is available at: <https://www.home-assistant.io/integrations/zha#compatible-hardware>. *Figure 5.1* shows one of the coordinator devices I used with Z2M application.



Figure 5.1 – Z2M Zigbee coordinator

Zigbee router

A Zigbee router is typically any mains-powered Zigbee device, such as a smart plug, smart switch, or smart bulb. It can also be a dedicated dongle or standalone device designed only to extend the Zigbee network without providing any smart home functionality. Zigbee routers act as repeaters, relaying messages between devices and creating the mesh backbone. They play an important role in extending network coverage and automatically repairing communication paths if another router fails.

In small homes or apartments with only a few nearby devices, a Zigbee router may not be necessary because the coordinator can communicate directly with end devices that are within radio range. However, as you add mains-powered Zigbee devices to your home, most of them automatically function as routers, allowing the mesh to grow naturally and improving network coverage and reliability.

An example of a Zigbee router is presented in *Figure 5.2*. This smart plug has router capabilities.



Figure 5.2 – Zigbee router example

Zigbee end device

End devices are any battery-powered Zigbee device connected to your network. They are low power and do not carry traffic. They sleep frequently, which is important for battery life, and wake up when an event happens, such as a door opening or a person being detected. Examples of these devices are motion sensors, door sensors, and buttons.

An example of a Zigbee end device is presented in Figure 5.3.



Figure 5.3 – Zigbee end device example

Mesh network

One of the powerful concepts used by Zigbee is the architecture network known as Mesh. This network is composed of different devices mentioned in the previous subsection. These devices, also called nodes, have the capability to communicate with each other and to create communication paths that can vary according to how they are distributed in the environment. Each communication between nodes is a hop, and Zigbee uses a multi-hop protocol where the best communication path is created based on a Link Quality Indicator (LQI).

Mesh communication paths, or routes, can be modified if a device is added or removed from the network, or if the LQI of one or more devices changes. The LQI is evaluated periodically. A

good placement of routers can improve considerable network performance. We will see how to better place devices at home later in this chapter.

At the time that I'm writing this chapter, my Zigbee network map is presented in *Figure 5.4*. This map was obtained using Zigbee2MQTT application for Home Assistant.

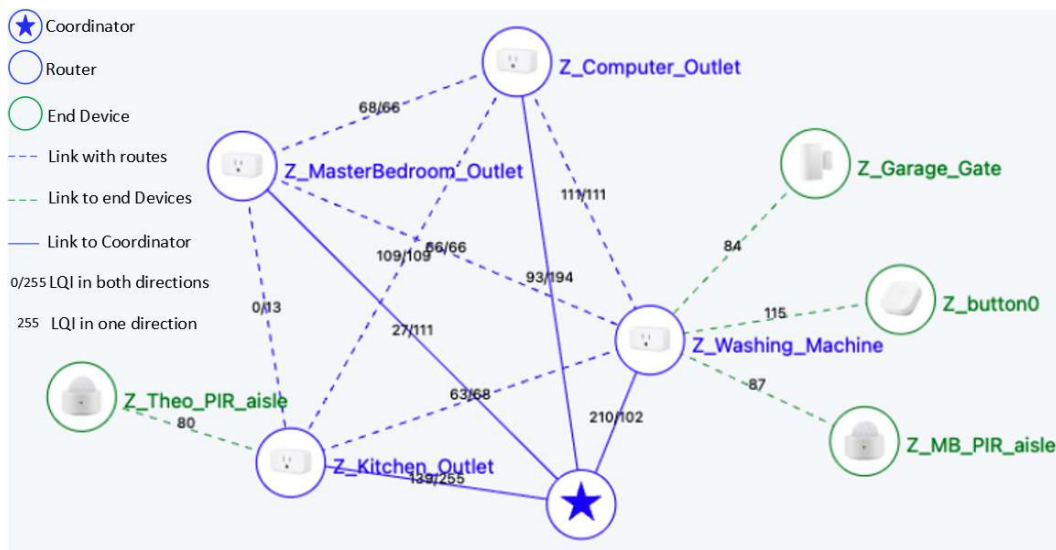


Figure 5.4 – My Zigbee network map using Zigbee2MQTT

As you can see in *Figure 5.4*, all of my Zigbee devices are named starting with the capital letter Z to identify that they are Zigbee devices.

Practical field experience from Zigbee users and developers suggests maintaining roughly one Zigbee router for every 4–5 battery-powered devices to ensure good routing performance. Also ensure at least one router per room, and avoid installing them behind TVs, fridges, or metal enclosures. We will comment about some practical approaches to improve or keep good Mesh performance in a future section in this chapter.

Communication channels

Zigbee communication channels span the 2.4GHz frequency range, the same as Wi-Fi. It has 15 channels, numbered 11 to 26, each with 2 MHz of bandwidth. Channel 11's center frequency is 2405 MHz, while channel 26's is 2480 MHz.

The fact that Zigbee uses the same frequency range as Wi-Fi causes overlapping of channels and if they are not well selected can lead to interference promoting packet loss, pairing failures or poor mesh stability. *Figure 5.5* shows where this interference happens more often between

these two wireless communication protocols, assuming a bandwidth of 20 MHz for Wi-Fi channels.

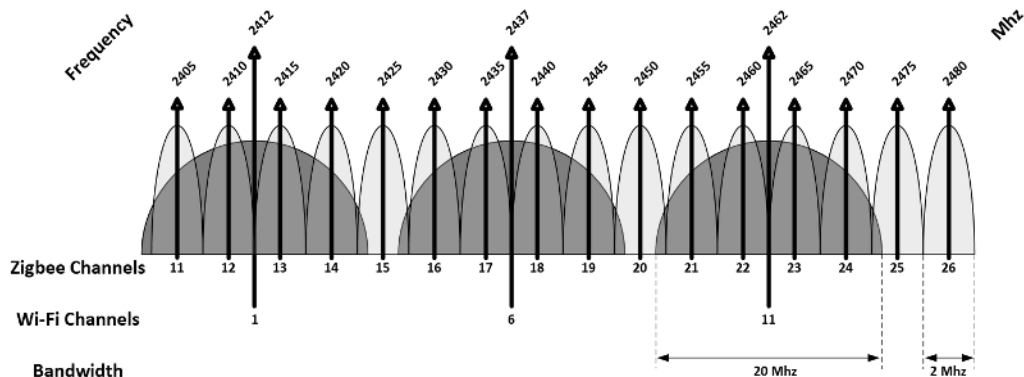


Figure 5.5 – Zigbee vs Wi-Fi channels overlapping

A good approach to avoid interference is to choose channels 6 or 11 for Wi-Fi and channels 11 to 15 for Zigbee. Also, by choosing Zigbee channels 15, 20, or 25, you will have the least overlap regardless of the Wi-Fi channel configured. Other configurations could be:

- Wi-Fi channel 1: Zigbee channels 20 or 25
- Wi-Fi channel 6: Zigbee channel 15
- Wi-Fi channel 11: Zigbee channel 20

Although Zigbee channel 26 operates at the upper edge of the 2.4 GHz ISM band and completely avoids Wi-Fi interference, it is less commonly used in practice. Many Zigbee devices transmit at reduced power on this channel to meet regulatory requirements, which can significantly reduce range and mesh reliability. In addition, some devices and routers have limited or inconsistent support for channel 26 due to RF and firmware constraints. For these reasons, channel 25 is often preferred, as it offers similar Wi-Fi isolation with better compatibility and overall network stability.

Note

If you want to check how congested the 2.4 GHz frequency spectrum around your home is, use an app like Wi-Fi Analyzer to evaluate it.

This section built the baseline knowledge necessary for designing robust Zigbee networks. By understanding device roles, mesh routing behavior and communication channels you now have the information required to plan your network topology before using the Zigbee applications we will start to see in the next section.

Zigbee2MQTT: advanced configuration and device management

There are two main applications you can use to manage Zigbee networks and devices in Home Assistant: Zigbee Home Automation (ZHA) and Zigbee2MQTT (Z2M). Both allow Home Assistant to communicate directly with Zigbee devices without depending on a vendor-specific bridge or gateway, but they follow different approaches. ZHA is built into Home Assistant and is usually simpler to start with, while Z2M requires a few more configuration steps and depends on MQTT as the communication layer.

We will begin with Zigbee2MQTT because its setup exposes more of the Zigbee network structure, including coordinator firmware, MQTT communication, network parameters, pairing behavior, and device management. Understanding these elements first will make it easier to recognize what is happening behind the scenes when using ZHA later. Z2M is also one of the most flexible Zigbee integrations available for Home Assistant, especially for users who want advanced device support, detailed diagnostics, and the ability to reuse Zigbee data in other platforms that support MQTT.

In this section, we will discuss the main aspects of installing and configuring Zigbee2MQTT, including firmware flashing, network configuration, device pairing, and additional features that can help you manage and troubleshoot your Zigbee mesh.

Z2M installation and firmware flashing

You can install and configure the Zigbee2MQTT Home Assistant app by following these steps:

1. In Home Assistant, go to Settings | Apps.
2. Click Install app.
3. Search for and select Zigbee2MQTT.
4. Install the Zigbee2MQTT app.
5. Connect your USB Zigbee dongle to your Home Assistant server. *Figure 5.1* shows an example of a USB Zigbee dongle.
6. Check which firmware version is currently installed on the dongle. If the firmware is outdated, it may not work correctly with some Zigbee devices, so you may need to flash a newer firmware version.
7. To flash new firmware, connect the dongle to your computer.
8. Install the appropriate USB drivers for your operating system and dongle model.
9. Open your browser and go to the firmware flashing tool at: <https://darkxst.github.io/silabs-firmware-builder/>

10. Click Connect and select your Zigbee dongle.
11. Download the most recent firmware file for your dongle. The firmware file usually uses the .gbl file extension. At the time of writing, the most recent version used in this chapter is NCP_7.4.4, available at: https://github.com/itead/Sonoff_Zigbee_Dongle_Firmware/tree/master/Dongle-E/NCP_7.4.4
12. Use the flashing tool to upload the .gbl firmware file to the dongle.
13. After flashing is complete, reconnect the dongle to your Home Assistant server.
14. In Zigbee2MQTT, go to Settings | MQTT.
15. Configure the MQTT broker credentials, including the server address, username, and password.
16. Go to Settings | Serial.
17. Configure the Port parameter. Depending on your installation, this may be similar to one of the following: /dev/serial/ttyUSB0 or: /dev/serial/by-id/usb-Itead_Sonoff_Zigbee_3.0_USB_Dongle_Plus_V2_dongle-number_here
18. In the same Serial tab, configure the Adapter field as: ember
19. Click Submit to confirm your choices.

After completing these steps, Zigbee2MQTT should be installed, the coordinator firmware should be updated if required, and the app should be ready to communicate with your MQTT broker and Zigbee coordinator.

Network configuration and pairing devices

Based on the channel recommendations given in the last section, configure the Zigbee communication channel in the Z2M settings | Advanced tab.

After installation and firmware flashing, you will have to set the Permit join (All) at the top of the app. When you do that, the network is open to add new devices for 4–15 minutes. Z2M automatically performs an interview, where it queries the device for clusters and capabilities, battery levels, supported attributes, manufacturer information and binding capabilities. If everything goes right with your devices connecting to Zigbee network you should have a dashboard like *Figure 5.6*.

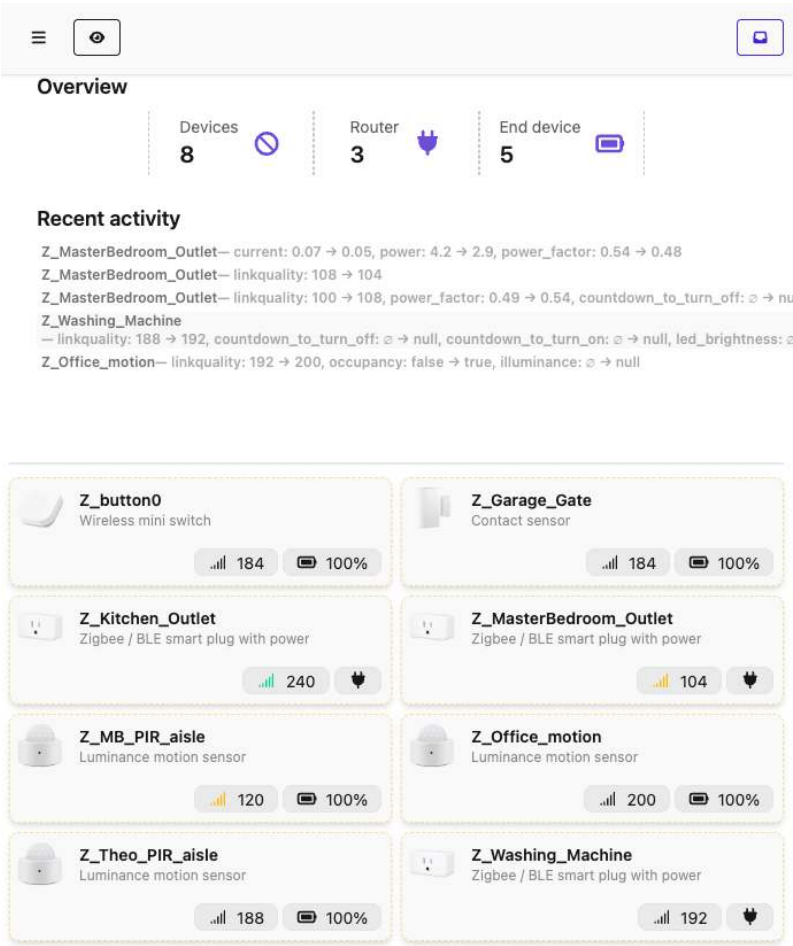


Figure 5.6 – Z2M dashboard

Binding, link quality metrics, OTA and groups

Zigbee networks provide a feature known as binding. This feature links two Zigbee devices directly to the Zigbee layer without the need to have Home Assistant involvement. The binding benefits are faster response, network traffic reduction, and it works even if Home Assistant is down. You can have a remote button binding to a bulb, plug or switch for example. In Z2M you can configure this feature by clicking on the Device and then Bind.

Another feature that Zigbee networks present is Groups. Using this feature, one command can control multiple devices. It provides an efficient broadcast message system reducing the latency for scenes. You can configure groups in the main menu using the option Groups.

Z2M offers a good visualization of the Link Quality metrics. The Z2M network map presented in *Figure 5.4* can be used to diagnose weak links.

Over-the-Air (OTA) is the capability for a device's firmware to be updated remotely, in this case using the Zigbee network. Z2M offers this feature for many vendors and periodically checks for updates, downloading them from remote repositories. You can control the OTA for devices using the OTA option in the main menu.

Zigbee2MQTT provides a powerful, flexible, and friendly way to manage a Zigbee network in Home Assistant. In the next section, we will explore ZHA, Home Assistant's native Zigbee integration, comparing its architecture, workflow, and capabilities to understand when it may be the better choice than Z2M.

Zigbee Home Automation (ZHA): setup and optimization

ZHA is Home Assistant's native Zigbee integration. It is typically the first Zigbee application chosen by new and non-technical users. In this section you will learn how to set up ZHA, optimize network behavior, manage devices effectively, and understand when ZHA is the right choice compared to Zigbee2MQTT. It is completely described at <https://www.home-assistant.io/integrations/zha>.

Zigbee Home Automation installation and device pairing

You can install and configure the ZHA integration in Home Assistant by following these steps:

1. In Home Assistant, go to Settings | Devices & Services.
2. Click Add Integration.
3. Search for ZHA.
4. Select Zigbee Home Automation from the integration list.
5. Select the Zigbee coordinator you want to use.

In my case, I used another Zigbee coordinator called Dongle Lite MG21 from Sonoff, shown in *Figure 5.7*, connected to a secondary Home Assistant server used for testing.



Figure 5.7 – My ZHA Zigbee coordinator

1. Follow the configuration wizard to complete the integration setup.

2. After the integration is installed, start adding Zigbee devices by clicking Add device inside the ZHA integration.
3. Put your Zigbee device into pairing mode. On most devices, pairing mode is activated by pressing and holding a button for a few seconds until an LED begins flashing or blinking, although the exact procedure and visual indication vary by manufacturer.
4. Wait for Home Assistant to discover the device and complete the pairing process.

After devices are attached to the network, ZHA provides configuration options similar to those available in Zigbee2MQTT. To access them, follow these steps:

1. Go to Settings | Devices & services.
2. Select the Zigbee Home Automation integration.
3. The main ZHA integration screen will be displayed, as shown in *Figure 5.8*.

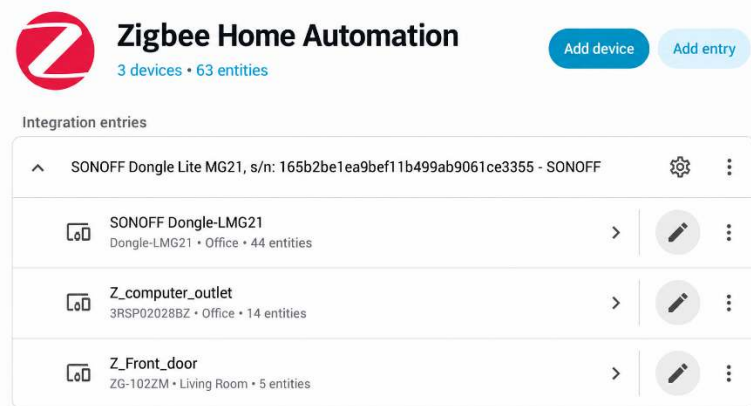


Figure 5.8 – ZHA integration main screen

4. Click the gear icon shown in *Figure 5.8*.
5. You will see three main tabs:
 - **Network:** Allows you to configure network settings, such as the Zigbee channel number.
 - **Groups:** Allows you to create Zigbee groups, like Zigbee2MQTT.
 - **Visualization:** Creates a network map for the Zigbee devices installed in your home.

Compared to Z2M, ZHA has fewer and more simplified configuration options. In the previous sections, you learned how Z2M exposes more advanced settings related to MQTT, serial communication, firmware, pairing, and device management, while ZHA provides a more

integrated and streamlined setup directly inside Home Assistant. This difference does not make one option better than the other, but it reflects two different approaches to managing Zigbee networks. In the next subsection, we will compare ZHA and Z2M more directly so you can decide which integration best matches your own setup, technical comfort level, and goals.

Zigbee Home Automation optimization

Both Zigbee Home Automation and Zigbee2MQTT use the same Zigbee protocol, but they have different abstraction layers inside Home Assistant.

ZHA is part of Home Assistant core, so it shares the event bus, state machine, and device registry structures explained in *Chapter 1*. It relies on stable Zigbee stacks like EZSP used in the G21 and G24 chipsets. The optimization happens by letting the Zigbee stack manage routing automatically and using Home Assistant concepts like areas and groups. The application simplifies the configuration for the user by minimizing the number of parameters to be configured. In other words, ZHA optimizes by its design, not by configuration. The optimization happens inside Home Assistant.

On the other hand, in Zigbee2MQTT, the Zigbee stack-level optimization is done outside Home Assistant via an app. Zigbee2MQTT runs a separate Zigbee controller application, exposing low-level Zigbee parameters and using MQTT as a transport layer. The optimization in this case happens by manually tuning reporting intervals, editing device-specific YAML configurations, forcing binding, creating groups, and managing OTAs and coordinator firmware. So Z2M optimizes the Zigbee network by configuration and not by abstraction.

As can be seen, there are practical trade-offs between Z2M and ZHA. We will discuss this a bit more in the next subsection.

Comparison between ZHA and Z2M: when to use one or the other

When it comes to ease of setup, ZHA is the simplest option. You enable it directly from the Home Assistant interface, without installing extra services or configuring MQTT. This makes it quick to deploy and easy to maintain. Z2M requires more initial work, including setting up an MQTT broker and managing an additional service. That extra effort gives you more control, but it also increases setup time and complexity.

For stability and maintenance, ZHA follows a conservative approach. It hides most low-level settings and relies on stable Zigbee stacks, which means you usually do not need to adjust anything after the initial setup. Z2M exposes many configuration options and evolves faster. You can fine-tune your network, but you also need to pay more attention to updates and configuration changes to keep things stable.

The biggest difference appears in device support depth. With ZHA, you get features that map cleanly to Home Assistant entities and follow the Zigbee specification. Manufacturer-specific or experimental features are often hidden. Z2M exposes more device options, including vendor-specific features and newly released devices. If you use less common hardware or want full access to device capabilities, Z2M usually gives you more.

In terms of debugging, observability, and long-term reliability, ZHA keeps things simple. You see clear entity states and basic network information, but little detail about raw Zigbee traffic. Z2M gives you deep visibility through logs, MQTT topics, and attribute-level data, which helps when troubleshooting complex issues. Over time, ZHA tends to be more predictable, while Z2M gives you more flexibility to adapt your network as your setup grows.

Table 5.2 summarizes a feature comparison we just made between both applications.

Feature	ZHA	Z2M
Ease of setup	Very simple	More complex
Maintenance effort	Low	Moderate
Device support	Conservative	Extensive
Debugging tools	Basic	Advanced
Long-term behavior	Stable and predictable	Flexible and adaptable

Table 5.2 – ZHA vs Z2M features comparison

In summary, if you want a Zigbee network that is easy to set up and mostly hands-off, ZHA is a good choice. If you want deeper control, better device support, and detailed troubleshooting tools, Z2M is usually the better option.

In the next section we will move beyond software configuration and focus on the physical design of the Zigbee network. You will learn how router placement, signal propagation, and network density affect mesh quality and reliability. This will help you build a Zigbee network that remains stable and responsive as it scales.

Building a robust mesh: router placement and network health

In this section, you will learn how to design, evaluate, and improve a Zigbee mesh by placing the coordinator correctly, selecting high-quality routers, and validating network health using

real metrics such as LQI, RSSI, and routing depth. The goal is to eliminate blind spots, reduce retries, and ensure consistent device responsiveness.

Coordinator placement best practices

The Zigbee coordinator is the center of the mesh, and all routing decisions depend on how well it can communicate with nearby routers. If the coordinator has poor signal conditions, the entire network becomes unstable, with slow responses and frequent dropouts. This is why correct placement is more important than adding extra routers.

Always connect the coordinator using a USB extension cable of at least 1 to 2 meters. Plugging it directly into the host exposes it to electrical noise that reduces radio performance. Keep the coordinator at least 1–2 meters away from Wi-Fi routers, access points, and USB 3.0 devices to minimize interference in the 2.4 GHz band.

Place the coordinator in a central and elevated location, such as on a shelf, and avoid basements, metal cabinets, or areas near large appliances. Dense materials like concrete walls, electrical panels, and aquariums weaken Zigbee signals. A poorly placed coordinator cannot be fixed by adding more routers, so this should always be the first step when building a reliable Zigbee network.

Router selection: choosing the right Zigbee repeaters

Routers are the backbone of a Zigbee mesh, as they carry traffic between the coordinator and end devices. Not all routers perform equally, and low-quality routers can reduce stability instead of improving it. Choosing reliable routing devices is essential for a healthy and scalable network.

Smart plugs are the most reliable option and are the most commonly recommended. Because they are always powered, they offer consistent routing and usually have good RF performance, making them ideal for hallways, living rooms, and open spaces. In-wall switches and dimmers are also useful, as they provide permanent routing points, but their performance varies depending on the manufacturer and antenna design. Dedicated Zigbee repeaters can help fill coverage gaps, especially in areas without accessible outlets or switches.

Battery-powered devices should never be used as routers as they do not forward traffic. Cheap Zigbee bulbs acting as routers and older Zigbee Home Automation 1.2 devices with limited memory can also destabilize the mesh.

Determining the right number of routers for a Zigbee network

A Zigbee network needs enough routers to keep routing paths short and reliable. As mentioned before, a practical rule of thumb is to use one router for every four to five end devices, with at

least one router in each major room or hallway. In multi-story homes, each floor should have a minimum of one router to ensure proper vertical coverage.

Following this ratio helps reduce the number of hops between devices, allows the mesh to recover faster when devices are added or removed, and lowers packet retries and latency. *Table 5.3* lists estimated Zigbee router density guidelines based on typical residential deployments.

Home Size	Approx. Area (sqft / m ²)	End Devices	Recommended Routers
Apartment	≤ 900 sqft / ≤ 85 m ²	~15	4–5
Small house	900–1,800 sqft / 85–170 m ²	~30	7–8
Large house	≥ 1,800 sqft / ≥ 170 m ²	50+	12+

Table 5.3 – Practical Zigbee router density for residential environments

The recommendations in *Table 5.3* are based on Zigbee deployments, device vendor guidelines, and community best practices observed across Home Assistant, Z2M, and ZHA installations. Also, these values in *Table 5.3* assume typical drywall construction.

Homes with concrete walls, metal framing, or multiple floors may require additional routers. As a practical starting point, consider adding at least one router near each major obstruction or on each floor, then use LQI, RSSI, and the Zigbee network map to determine whether additional routers are needed.

Diagnosing Zigbee signals with LQI and RSSI

Z2M and ZHA expose diagnostics that make it possible to evaluate the health of a Zigbee network. The most useful indicators are LQI, RSSI, and routing depth, which together show how stable and efficient each device's connection is. *Figure 5.9* shows LQI and RSSI connection levels for Zigbee networks.

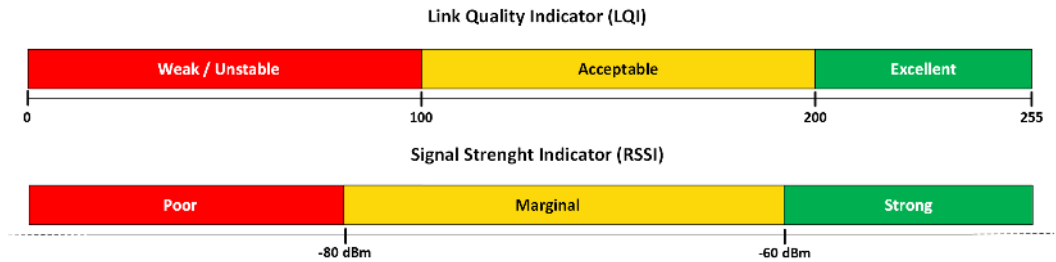


Figure 5.9 – LQI and RSSI values

LQI (Link Quality Indicator) represents the relative quality of a Zigbee link on a scale from 0 to 255. Values above 200 indicate an excellent connection, values between 120 and 199 are generally acceptable, and values below 100 often point to an unstable or weak link. RSSI measures signal strength in dBm, where values between -40 and -60 dBm are strong, around -70 dBm are marginal, and values below -80 dBm usually lead to unreliable communication.

It is important to mention that LQI and RSSI are not equivalent. RSSI measures raw signal strength in dBm, while LQI represents the overall quality of a Zigbee link, including the effects of noise, interference, and packet errors.

In addition to the link quality and signal strength, routing depth is important because most devices should ideally connect within three hops or fewer to avoid latency and packet loss.

Coverage gaps can be identified using the network visualization tools available in Zigbee2MQTT and ZHA. These tools display the mesh topology, showing how end devices connect to the coordinator through routers while also exposing LQI and RSSI metrics. Areas containing devices with consistently low LQI values, weak RSSI, or long routing paths often indicate poor coverage and are good candidates for repositioning an existing router or adding another mains-powered Zigbee router.

When diagnosing issues, start by identifying devices with low LQI values or unusually high hop counts. Check which router those devices are using and whether a closer or better positioned router is available. Adding a router along the path or repositioning an existing one often improves stability. In some cases, re-pairing the device helps it select a better route through the mesh.

We will see in the next subsection more about diagnostics and how to evaluate mesh improvements after changes in some devices location.

Identifying poor-quality devices and evaluating mesh improvements

As mentioned before, Zigbee routers can contribute negatively to the mesh, and some devices can silently degrade network stability. Common warning signs include many end devices attaching to a single router and frequently dropping offline, devices choosing distant routing paths even when closer routers are available, and random offline events that occur without any power interruption. These symptoms often point to routers with poor RF performance, limited buffering, or firmware issues that disrupt normal mesh behavior.

When these issues appear, corrective action should be done step by step. Start by removing the suspected router and observing how the mesh reorganizes over time. If stability improves, replace the device with a known-good router and re-pair any end devices that were previously

routing through it so they can select better paths. In many cases, Zigbee meshes show immediate and noticeable improvement after a single problematic router is removed.

A practical way to confirm whether mesh changes are effective is to run a simple, controlled evaluation. You can perform an **exercise** on an existing Zigbee network to verify improvements. It allows you to observe, with real data, how small physical changes impact mesh behavior and stability. Follow the sequence below to do that:

1. Record baseline values for LQI, RSSI, and hop count for a few selected devices that show weak or unstable behavior.
2. Move the coordinator using a USB extension cable to a central or elevated position.
3. Add a Zigbee router in an area with poor coverage or frequent dropouts.
4. Allow the mesh to reconfigure naturally, or re-pair affected devices to force route recalculation.
5. Compare the new metrics with the baseline and observe changes in device responsiveness and stability.

After completing this exercise, you should see shorter routing paths, more consistent LQI values, and faster device response times, confirming that coordinator placement and router density have a direct and measurable impact on Zigbee mesh health.

In practice, it is not always possible to position the coordinator or routers exactly as recommended in this section. Home layout, wall materials, electrical outlet locations, and existing installations often impose constraints that limit ideal placement. Even so, understanding how placement and router quality affect the Zigbee mesh allows you to make informed compromises and apply improvements where they have the greatest impact.

In the next section, you will move from optimization guidelines to a complete hands-on project. This practical example implemented in my home applies the concepts covered in this chapter showing how Zigbee and Wi-Fi devices can be integrated into a unified and reliable automation solution.

Note

Z-Wave: An Alternative Protocol

Before we move to the next section, let's talk a bit about Z-Wave.

Z-Wave is a strong alternative to Zigbee, especially in environments where radio interference is a concern. Its main advantage is that it operates on sub-GHz frequencies, instead of the crowded 2.4 GHz band used by Zigbee and Wi-Fi. Depending on the region, Z-Wave typically runs at 908.42 MHz in North America and 868.42 MHz in Europe, which significantly reduces interference from Wi-Fi/Zigbee networks and

Bluetooth devices. This lower frequency also allows Z-Wave signals to penetrate walls and floors more effectively, making it particularly attractive for larger homes or installations with dense construction materials.

The main drawbacks of Z-Wave are cost and availability. Z-Wave devices are more expensive than their Zigbee equivalents, and the ecosystem offers fewer device models and vendors. Additionally, Z-Wave requires certified hardware, which limits experimentation and increases entry cost. To address some of these barriers, Nabu Casa introduced the ZWA-2 in August 2025, a Z-Wave coordinator designed specifically for Home Assistant. This initiative improves integration quality and long-term support, making Z-Wave a more approachable option for users who value reliability and reduced RF interference over device variety and price.

Hands-on project: integrating Wi-Fi and Zigbee for garage gate control

After covering Home Assistant architecture, advanced YAML, templates, Zigbee networking, and smart home network design, this chapter concludes with the first hands-on project of the book. The project brings together several concepts introduced so far and demonstrates one of the most powerful aspects of Home Assistant: integrating hardware, software, and different communication technologies into a unified automation system...

This project is included in the Zigbee chapter because it demonstrates how different wireless technologies can coexist and complement each other. Zigbee is well suited for low-power input devices, such as battery-powered buttons, while Wi-Fi provides flexibility for custom hardware control using ESP8266 or ESP32-based boards.

In this project, you will use a Zigbee battery-powered button to control a garage gate through a Wi-Fi relay activated by an ESP8266 module running Tasmota. Home Assistant will connect both sides of the system through an automation. In the next subsections, you will review the hardware architecture, mechanical considerations, electronics connections, firmware configuration, and Home Assistant automation required to complete the solution.

Zigbee garage gate control overview

The first thing to describe is the components of this hands-on project. These components are presented in *Figure 5.10*.

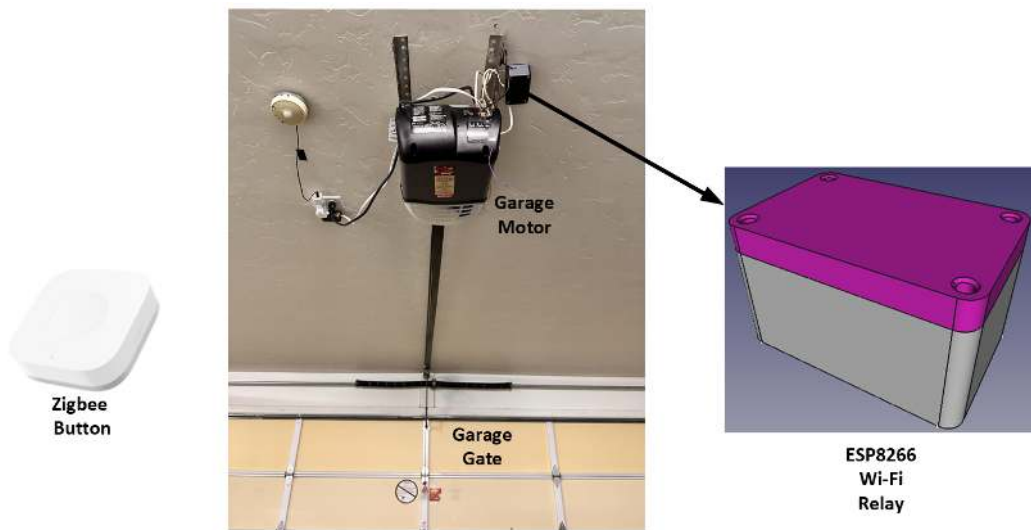


Figure 5.10 – Zigbee garage door components

From left to right in *Figure 5.10* we have:

- Zigbee button: battery powered Zigbee button. In my case, I'm using an Aqara model WXKG11LM, but you can use whatever Zigbee button you want.
- Garage Motor: regular Garage Motor attached to the garage gate. It has a light curtain sensor that stops the motor in case it is crossed. These motors are common, at least in the USA. The one installed in my garage is a Craftsman model 41AC150-1M.
- ESP8266 Wi-Fi Relay: This is an ESP8266 Wi-Fi Module assembled on a board called LC-Relay-ESP12-1R-MV. We will describe this board in more detail later.

Behind the scenes, and not shown in *Figure 5.10*, is the Home Automation server — in my case, a Raspberry Pi 4 with Home Assistant installed, connected to Wi-Fi. The ZBDongle-E Zigbee coordinator presented in *Figure 5.1* is attached to the Raspberry Pi USB port.

The way the solution works is that once the Zigbee button is pressed, it triggers a Home Assistant automation that toggles the Wi-Fi board relay, emulating the physical button connected to the garage motor. The Garage Motor then opens or closes the garage gate depending on its state. Simple as that.

In the next subsections, I will explain details about mechanics and fast prototyping, electronics and the software as part of the solution.

Mechanics and fast prototyping

Because the ESP8266 Wi-Fi relay module must be installed close to the garage motor, it should be placed inside an enclosure to protect the board and keep the installation organized. A 3D-printed enclosure is an ideal way to house the ESP8266 board safely near the garage motor, especially when the board dimensions or mounting requirements do not match a standard off-the-shelf box.

For this project, the enclosure was designed as a rectangular 3D-printed box with a separate lid using FreeCAD (www.freecad.org), a free and open-source 3D modeling tool. *Figure 5.11* shows the enclosure design, with the box shown in gray and the lid shown in magenta.

The design uses a parametric approach, which means the enclosure dimensions can be adjusted by changing values in the spreadsheet section of the FreeCAD file. This makes the model reusable for other boards or future smart home projects that require custom enclosures. The FreeCAD design file and the 3D printer files are available in the book's GitHub repository. Please refer to the *Preface* of the book to get the link to the repository.

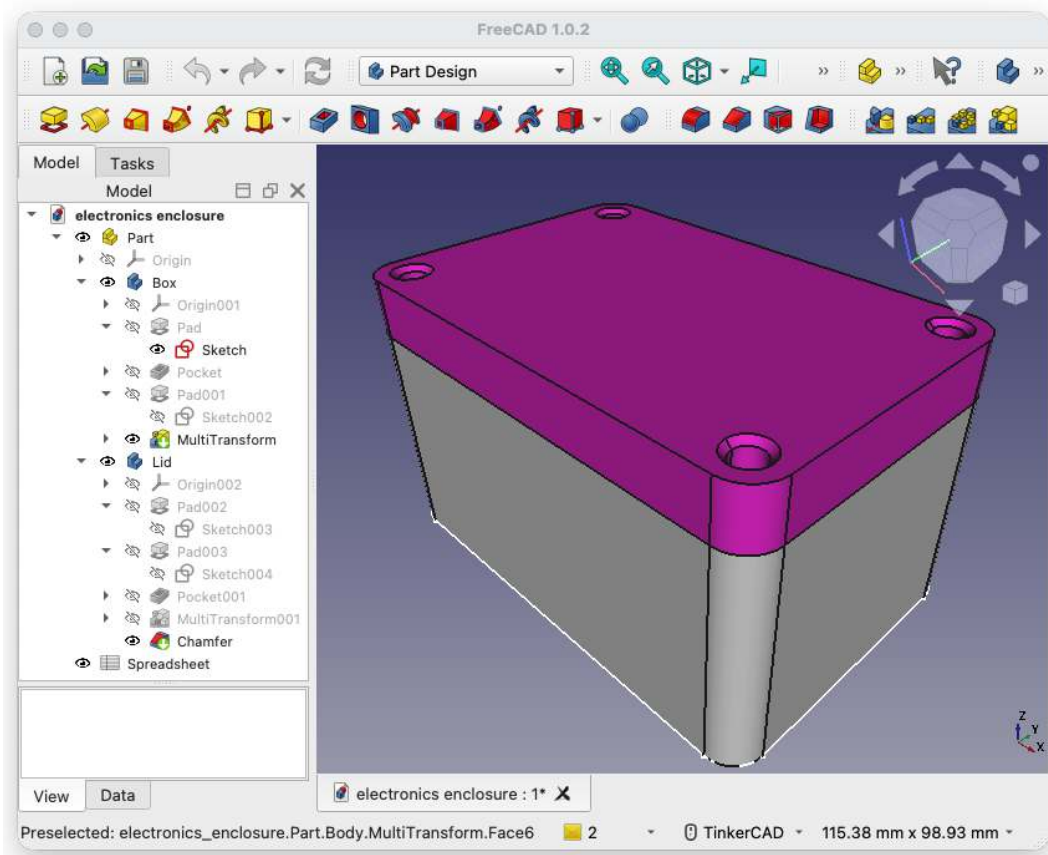


Figure 5.11 – Electronics enclosure box and lid developed using FreeCAD

After the box and lid were designed, they were both printed at the same time. The 3D print process and the electronic board fitment check are presented in Figure 5.12.



Figure 5.12 – Box with lid being printed and electronic board fitment check

The box fitment worked perfectly. The design included two openings on both sides of the box, to connect it to the AC voltage and the relay output. The box and lid were assembled using four screws that pass through the top of the lid and screw into the bottom of the box, holding the electronic board in place through its four mounting holes.

Electronics board

As mentioned before the board used in this project is called LC-Relay-ESP12-1R-MV (<http://www.chinalctech.com/m/view.php?aid=472>) presented in *Figure 5.12* on the right. It is an 80mm x 50mm board based on the ESP-12F Wi-Fi Module which has the ESP8266 chip. It has a single relay with 10A 250/125VAC specifications. It also has an input able to manage 90-250VAC, which means I can connect the board straight to the AC voltage line.

This board is the perfect solution for this application, but if you can't find it, you can also use a single-output smart switch that will do the same job. The LC-Relay-ESP12-1R-MV has all ESP8266 pins exposed, and you can use jumper wires to connect to other sensors if required.

Another advantage of this board is that, as it is based on the ESP8266, it supports Tasmota, which offers a template configuration that can be used with the board. For more information on it check this link: https://templates.blakadder.com/ESP12F_Relay_X1.html. This link will be useful for configuring Tasmota to run on this board.

Assuming you already installed Tasmota on the board and the MQTT server configured, to set up the pinout, follow these steps:

1. Open the Tasmota webserver using the IP address you got during the installation
2. Click on **Configuration | Template**
3. In the template parameters section, fill the field name as LC-Relay-ESP12-1R-MV and the field Based on as Generic (18).
4. In the GPIOs area, fill GPIO2 as LedLink, GPIO5 as Relay 1, and GPIO16 as Led_i 2.
5. Click on the **Save** button.

You will have to connect the board relay output according to *Figure 5.13*.

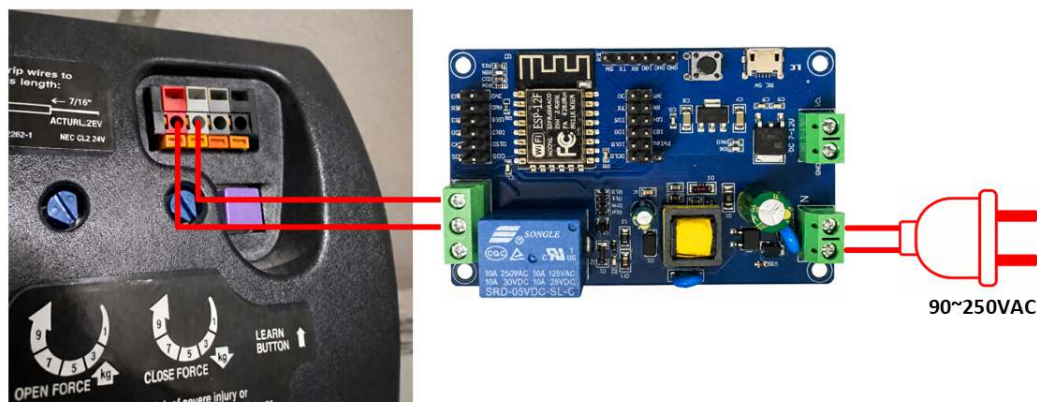


Figure 5.13 – Board to garage motor wiring diagram

The relay contacts marked Normally Open (NO) and Common (COM) on the board should be wired to the red and gray terminals in the garage motor control circuit. Each time the relay is activated, it will close the circuit connected to these terminals, and the garage motor will understand that it has to be closed or opened. *Figure 5.13* doesn't show it, but there are other wires connected to the garage motor connector. Do not remove these wires otherwise your light curtain sensor and the manual switch will stop working.

Note

Warning: Incorrect wiring can damage your hardware or create a safety hazard. Check your board and garage motor specifications before making any of these wiring changes. I do not take any responsibility for any damage to your board, garage motor, or yourself if something is wired incorrectly.

Next, we will create the automations to control the garage motor from the Zigbee button.

Zigbee garage motor automation

The final step of this hands-on project is to create a Home Assistant automation that will integrate all of the devices presented in *Figure 5.10*. It is important to name the devices, so the Zigbee button is called `Z_Button0` and the Wi-Fi board is called `GarageMotor`.

Before I created the Home Assistant automation, I realized that the Open/Close action for the garage motor could be used not only by the Zigbee button, but by Home Assistant itself and also to create other automations using my phone for example, closing the garage door each time that Home Assistant senses that I left home. I then created a script that could be reused by other automations or across Home Assistant.

Most residential garage door openers use a momentary push button to control the motor. Pressing the button briefly closes a low-voltage circuit, which the garage door controller interprets as an open or close command. To emulate this behavior, I created a Home Assistant script that turns on the GarageMotor board relay for 500 milliseconds and then turns it off. The 500 ms interval only simulates pressing the button; the garage door controller handles the complete door movement automatically. The flowchart for the Turn On/Off GarageMotor script is shown in *Figure 5.14*.

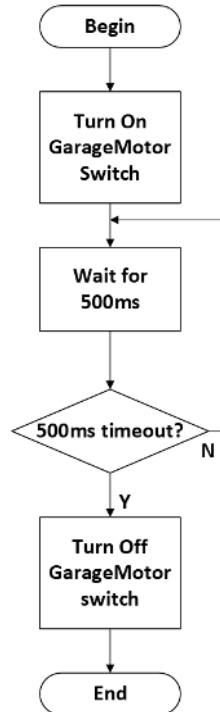


Figure 5.14 – Turn On/Off GarageMotor script

The Script YAML code is presented in the following sequence.

```
alias: Turn On/Off GarageMotor Script
sequence:
  - action: switch.turn_on
    metadata: {}
    data: {}
    target:
      device_id: your_garage_motor_id
  - delay:
```

```

    hours: 0
    minutes: 0
    seconds: 0
    milliseconds: 500
  - action: switch.turn_off
    metadata: {}
    data: {}
    target:
      device_id: your_garage_motor_id
description: "GarageMotor - Turn On - wait 500ms - Turn Off "
icon: mdi:garage-variant

```

Creating the script makes the Home Assistant automation simple. It just uses the Z_Button0 single action as a trigger and then runs the script we just described. The flowchart for the automation called Garage Gate Open/Close is presented in *Figure 5.15*.

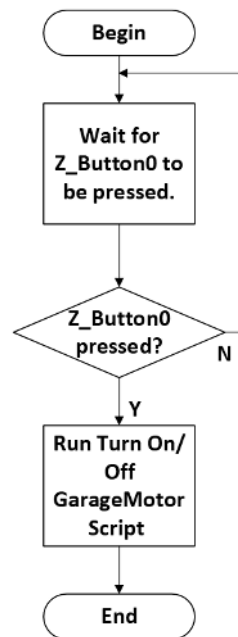


Figure 5.15 – Garage Gate Open/Close automation flowchart

The YAML code for the Garage gate Open/Close is the following:

```

alias: Garage Gate Open/Close
triggers:

```

```
- domain: mqtt
  device_id: your_zigbee_button_id
  type: action
  subtype: single
  trigger: device
conditions: []
actions:
- action: script.turn_on_off_garagemotor_script
  metadata: {}
  data: {}
mode: single
```

The implementation of this automation allows a Zigbee button to control the garage motor, opening or closing the garage gate through Home Assistant. This type of setup is useful when the garage gate needs to be controlled from another area of the house, such as the kitchen, without installing additional wired switches.

In this example, the Zigbee button triggers the ESP8266 relay module, which sends a dry-contact signal to the garage motor. A separate Zigbee contact sensor installed on the garage gate can be used to monitor whether the gate is open or closed. If the gate is left open, Home Assistant can alert the user and provide a convenient way to close it remotely by pressing the Zigbee button.

After the gate closes, a second automation can send a confirmation notification to the user's phone when the contact sensor reports the closed state. This creates a complete control-and-feedback loop: the Zigbee button initiates the action, the ESP8266 relay activates the garage motor, and the Zigbee contact sensor confirms the final gate position.

This hands-on project demonstrates that robust smart home automations are not about choosing a single protocol, but about combining the strengths of each technology. Zigbee provides efficient, battery-powered, reliable input devices, while Wi-Fi and Tasmota enable flexible, custom actuation hardware.

With this final project completed, you now have a practical reference design for building secure, scalable, multi-protocol automations that go beyond simple device control.

Summary

This chapter explains how to design, configure, and maintain a Zigbee mesh network using Home Assistant. It starts by introducing the Zigbee protocol, including device roles, mesh routing behavior, and channel selection, giving you the background needed to plan a reliable network.

The chapter then focuses on practical implementation using the two main Zigbee integrations available in Home Assistant: Zigbee2MQTT (Z2M) and Zigbee Home Automation (ZHA). You learn how to install, configure, and manage devices with each option, as well as the key differences between them. This helps you decide which integration best fits your setup and maintenance preferences.

A large part of the chapter is dedicated to mesh design and troubleshooting. You learn how coordinator placement, router quality, and router density affect network behavior, and how to use metrics such as LQI, RSSI, and hop count to evaluate network health. Common causes of instability are discussed, along with practical steps to improve reliability.

The chapter also briefly introduces Z-Wave as an alternative protocol, explaining when it may be a better choice due to reduced radio interference and what trade-offs it involves.

The chapter ends with a hands-on project that combines Zigbee and Wi-Fi devices to control a garage gate. This project shows how different wireless technologies can be integrated in Home Assistant using automations, custom hardware, and reusable scripts.

In *Chapter 6, Cloud IoT Integration with Golioth*, the focus moves beyond local networks to cloud-based IoT platforms. You will learn how services such as Golioth can be used to manage devices remotely, handle data securely, and integrate cloud-connected devices with Home Assistant, expanding your automation system beyond the local network.

6

Cloud IoT Integration with Golioth

So far, this book has focused heavily on local architecture, where Home Assistant runs as the central piece and devices communicate over local wireless networks like Zigbee and Wi-Fi. This approach provides low latency, privacy, and independence from external services.

However, not all devices can or should operate entirely within your local network. Many consumer IoT products rely on proprietary cloud services by design, and some applications genuinely benefit from cloud connectivity. Cloud platforms can provide capabilities that would be difficult or impractical to implement using only a local Home Assistant installation.

In this chapter, we will explore cloud IoT platforms you can manage and how they can complement a Home Assistant centralized architecture. You will learn when cloud connectivity is beneficial, how to securely connect devices to the cloud, and how to bridge cloud data back into Home Assistant without losing control of your automation logic.

The chapter focuses primarily on Golioth, a modern cloud platform designed specifically for embedded devices, while also providing a brief comparison with other cloud IoT solutions.

By the end of this chapter, you will understand the architecture and benefits of cloud IoT platforms, learn how to securely connect ESP-based devices to Golioth, integrate cloud-connected devices and data into Home Assistant, and evaluate different cloud IoT platforms to determine which best fits your home automation requirements.

In this chapter, we will cover the following topics:

- Cloud IoT platforms: benefits and use cases
- Golioth architecture and components
- Connecting ESP32 devices to Golioth

- Integrating Golioth Data and Devices with Home Assistant
- Exploring other cloud IoT platforms

Let's begin by exploring when and why cloud connectivity adds value to a Home Assistant-based smart home.

Technical requirements

No prior experience with cloud IoT platforms is required. However, to get the most out of this chapter, you should have the following.

Knowledge prerequisites:

- Familiarity with ESP32-based devices, flashing firmware, and reading serial logs.
- A solid understanding of basic networking concepts, including Wi-Fi connectivity, TCP/IP, HTTPS, and the difference between local and internet-facing devices.
- Basic knowledge of REST APIs, JSON payloads, authentication concepts, and event-driven communication.

Hands-on project requirements:

- A Golioth account (the free tier is sufficient).
- Visual Studio Code with the ESP-IDF extension installed and configured.
- A Wemos D1 R32 or another ESP32 development board compatible with ESP-IDF. A BMP280 temperature and pressure sensor module.

You can download the code for all examples in this book by following the instructions in the **Download the example code files** section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Cloud IoT platforms: benefits and use cases

Cloud IoT platforms extend the capabilities of a local Home Assistant installation by providing services that are difficult to implement efficiently within a single home network. Before exploring Golioth and other cloud platforms, it is useful to understand the situations where cloud connectivity adds value and how it complements a local-first smart home architecture.

Benefits of cloud connectivity in a smart home

A cloud IoT platform provides centralized services that are difficult or inefficient to implement in a local network especially if you need your system to grow beyond a single Home Assistant instance or physical location. These cloud platforms are designed to manage devices at scale, across unstable or changing networks, while maintaining security protocols.

Cloud IoT platforms offer several advantages that are difficult to replicate in a purely local network, including:

- **Device identity and provisioning:** Each device has a unique identity, credentials, and lifecycle managed centrally.

Note

Provisioning is the step that turns a physical device into a recognized and secure cloud-connected device.

- **Secure communication:** End-to-end encryption using TLS, certificates, and device authentication mechanisms.
- **Fleet-wide device management:** Monitor, configure, and manage a large number of devices consistently.
- **Over-the-air (OTA) firmware updates:** Safely deploy firmware updates without physical access.
- **Centralized telemetry storage:** Store large volumes of sensor data for long-term analysis and auditing.
- **Remote access without exposing your LAN:** Devices connect outbound to the cloud, avoiding port forwarding or VPN complexity.

Home Assistant manages local devices, automation logic, and user interface, but it was not designed to manage fleets of embedded devices deployed across multiple networks or locations. Cloud platforms complement Home Assistant by taking ownership of device connectivity, security, and lifecycle, while Home Assistant continues to be the local Home Automation Server.

Common use cases for cloud integration

Cloud IoT platforms support a wide range of applications beyond home automation. Although this book focuses on Home Assistant, understanding these broader use cases helps illustrate the flexibility and value of cloud connectivity. The hands-on examples use Golioth, which provides a free developer tier suitable for learning and experimentation. *Figure 6.1* illustrates the types of devices and entities that can connect to IoT cloud platforms and the different ways they interact with cloud services.

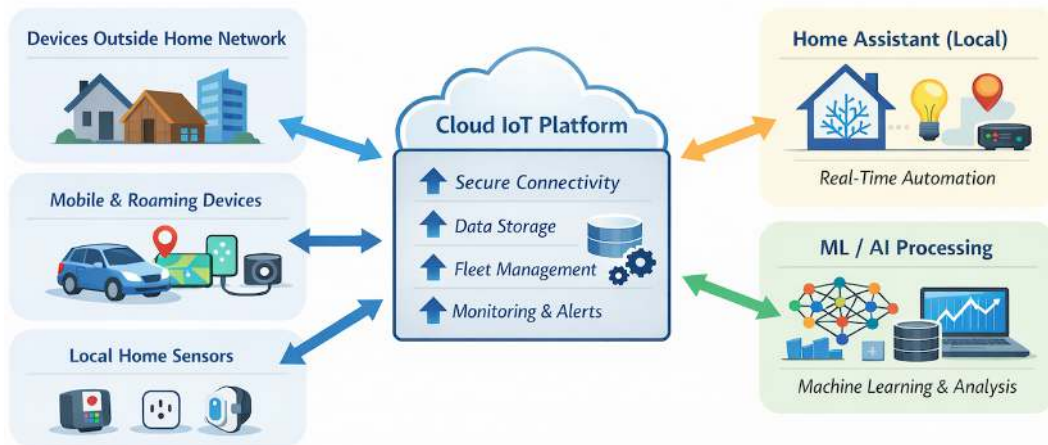


Figure 6.1 – Devices/entities and usage of IoT cloud platforms

Cloud IoT platforms become valuable in the following situations:

Devices outside your home network

Sensors deployed in a rental property, remote cabin, or a family member's home cannot reliably connect back to your local Home Assistant instance. Cloud platforms act as a common point, allowing data to flow securely regardless of network topology.

Mobile or roaming devices

Devices such as GPS trackers, vehicles, or portable sensors frequently change Wi-Fi or cellular networks. A cloud backend provides stable identity and connectivity, even when the network changes.

Long-term data retention

Storing months or years of high-frequency telemetry locally can fill the Home Assistant database and degrade performance. Cloud platforms are optimized for scalable storage and historical analysis, while Home Assistant can consume only the relevant data.

Fleet management

When managing multiple IoT devices, consistency becomes critical. Cloud platforms enable centralized firmware versioning, configuration rollout, health monitoring, and failure detection — tasks that are hard to scale using local networks.

Cloud-based data collection for machine learning and AI

Another important use case for cloud IoT platforms is collecting and storing data for machine learning (ML) or AI workloads, even when all devices are installed inside the home. Training

ML models often requires large, long-term datasets, consistent schemas, and processing capabilities that go beyond what is practical to run on a Home Assistant instance.

By streaming telemetry such as energy consumption, occupancy patterns, environmental data, or user interactions to the cloud, you can build historical datasets suitable for model training, validation, and experimentation. The cloud becomes a scalable data lake and compute environment, while Home Assistant continues to handle real-time automations and user interaction. This separation allows you to experiment with AI-driven insights without impacting the performance, stability, or storage constraints of your local system.

The key idea is not replacing Home Assistant but augmenting it by using IoT cloud platforms. Home Assistant remains the Home Automation engine and user interface, while the cloud can handle secure remote connectivity, scale, and device lifecycle management.

With these use cases in mind, the next section takes you from concept to implementation by introducing a concrete cloud IoT platform. You will explore Golioth's architecture and core features, and see how it supports secure device connectivity, data handling, and fleet management in practical IoT deployments.

Golioth architecture and components

Golioth is a cloud IoT platform designed for constrained embedded devices, such as ESP32-based hardware, Zephyr RTOS systems, and other microcontroller-class platforms. Instead of targeting large Linux-based gateways or enterprise-scale servers, Golioth focuses on the realities of embedded development: limited memory, limited processing power, and unreliable networks. It is an agnostic system, meaning it is not tied to any particular microcontroller or RTOS.

Unlike traditional enterprise IoT platforms that add unnecessary complexity, Golioth is designed with developers in mind. It focuses on simplicity, lightweight communication protocols, and security enabled by default. The platform follows clear design choices and predefined best practices, allowing devices to connect quickly. This makes Golioth well suited for makers, product developers, and engineers building reliable IoT devices that must be easy to deploy and maintain over time.

At its core, Golioth acts as a middleware between embedded devices and cloud services, handling identity, connectivity, data ingestion, configuration, and firmware lifecycle management so you can focus on device functionality instead of cloud configuration.

Core Golioth components

At a high level, Golioth is built around several tightly integrated components that work together to provide a complete device management and data platform. *Figure 6.2* presents the

main components and their relationships. In the hands-on example later in this chapter, *Figure 6.4* illustrates how several of these services interact in practice as an ESP32 device connects to Golioth, publishes telemetry, and receives cloud services such as configuration updates and OTA firmware updates.

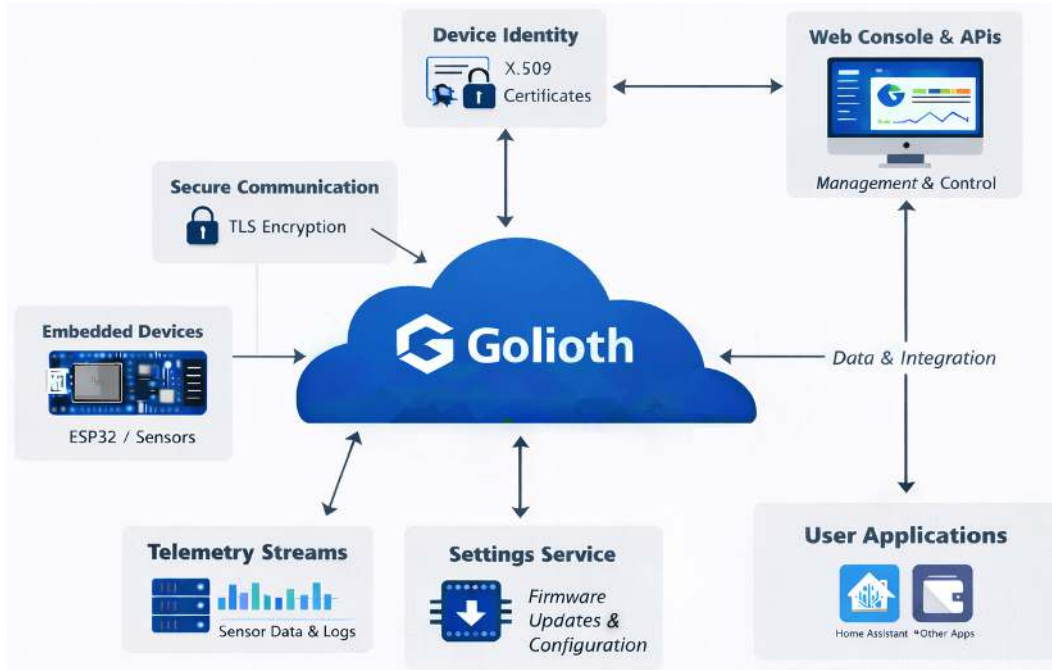


Figure 6.2 – Golioth platform components

The following subsections describe each component in detail, starting with the foundation of all Golioth communication: device identity.

Device identity

Every device connected to Golioth has a unique identity based on X.509 (<https://www.rfc-editor.org/rfc/rfc5280>) certificates. Each identity has a Pre-Shared Key Identification (PSK-ID) and a Pre-Shared Key (PSK) associated with it. This approach eliminates shared credentials and hardcoded secrets, ensuring that each device can be individually authenticated and revoked if necessary. Identity is established during provisioning and remains the foundation for all future communication.

Secure communication

All device-to-cloud communication uses encrypted channels based on Transport Layer Security (TLS). Security is a core design principle. From the first connection, data exchanged

between devices and Golioth is protected against interception and tampering, even when devices operate on untrusted networks.

Telemetry streams

Devices send structured telemetry data such as sensor readings, status updates, or diagnostics to Golioth using lightweight protocols optimized for embedded systems. This data can be stored, queried, forwarded, or integrated with external systems, while Home Assistant or other applications consume only what is relevant for automation and visualization.

Settings service

The settings service enables cloud-to-device configuration changes without requiring firmware reflashing. You can adjust parameters, thresholds, or feature flags remotely, allowing devices to adapt dynamically after deployment. This is especially valuable for tuning behavior in the field or managing fleets of devices.

OTA firmware updates

Golioth provides controlled over-the-air firmware updates with support for staged rollouts and rollback strategies. Devices can be updated incrementally, minimizing risk and downtime. This capability is essential for long term IoT deployments where physical access to devices is limited or impossible.

Web console and APIs

A centralized web console and well-documented APIs allow you to manage devices, inspect telemetry, configure settings, and automate workflows. These interfaces make it easy to integrate Golioth into Continuous Improvement/Continuous Development (CI/CD) pipelines, backend services, or Home Assistant, enabling a single view of both local and cloud-connected devices.

Embedded devices

The Embedded devices component represents the physical hardware that connects to Golioth. These are typically resource-constrained devices responsible for sensing, actuation, and local control. Golioth supports common microcontroller platforms such as the ESP32 family, Nordic nRF52/nRF91, and other ARM Cortex-M devices, making it suitable for both maker projects and production deployments.

From a software perspective, Golioth works with Zephyr RTOS, FreeRTOS, and other minimal RTOS or bare-metal environments when standard networking and TLS stacks are available. Devices run their core logic locally and communicate securely with Golioth for telemetry, configuration, and firmware updates, ensuring they remain functional even when cloud connectivity is temporarily unavailable.

Together, these components form a unified platform that hides much of the complexity involved in building and operating IoT systems. In the next section, you will put this architecture into practice by connecting an ESP32-based temperature sensor to Golioth and streaming telemetry data to the cloud.

Connecting ESP32 devices to Golioth

As discussed, one of the reasons for adopting a cloud IoT platform is when devices are deployed outside the local Home Assistant network. In this section, we explore how to connect an ESP32-based temperature sensor device to the Golioth IoT cloud and stream telemetry data from this sensor to it.

The use case application is for this sensor to be installed in a remote location, such as a vacation home, rental property, cabin, or family residence.

ESP32-based temperature sensor circuit

The example circuit used to test Golioth is based on the temperature sensor module BMP280 from Bosch. I connected the BMP280 breakout board to a Wemos D1 R32 board. The Wemos D1 R32 board is an Arduino Uno form factor board that uses an ESP32 WROOM 32 Wi-Fi Module. The interconnect diagram of the example circuit is shown in *Figure 6.3*.

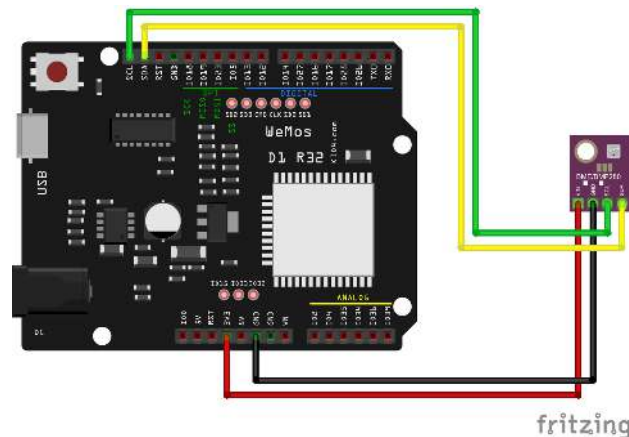


Figure 6.3 – ESP32 based temperature sensor – Golioth example

The wiring connection is straightforward between the two boards. The Vin power signal can be connected either on 3.3V or 5V. The next subsection provides details about the software portion of the example.

Firmware architecture overview

The firmware examples in this chapter use the Espressif IoT Development Framework (ESP-IDF), which is the development environment supported by Golioth for ESP32 devices. Because the examples rely on the Golioth Firmware SDK and ESP-IDF-specific configuration options, they cannot be implemented directly using the Arduino IDE. Install the ESP-IDF extension for Visual Studio Code and follow the setup instructions available at: <https://docs.golioth.io/getting-started/device-examples/compile-example-code/esp-idf/>.

Next, clone the Golioth Firmware SDK repository and configure the ESP-IDF development environment. After completing the setup, use the stream example available at: https://github.com/golioth/golioth-firmware-sdk/tree/main/examples/esp_idf/stream as the baseline for developing the temperature sensor application presented in this chapter.

The code created is available in the Book Repository and has the flowchart presented in *Figure 6.4* for the `app_main.c` file.

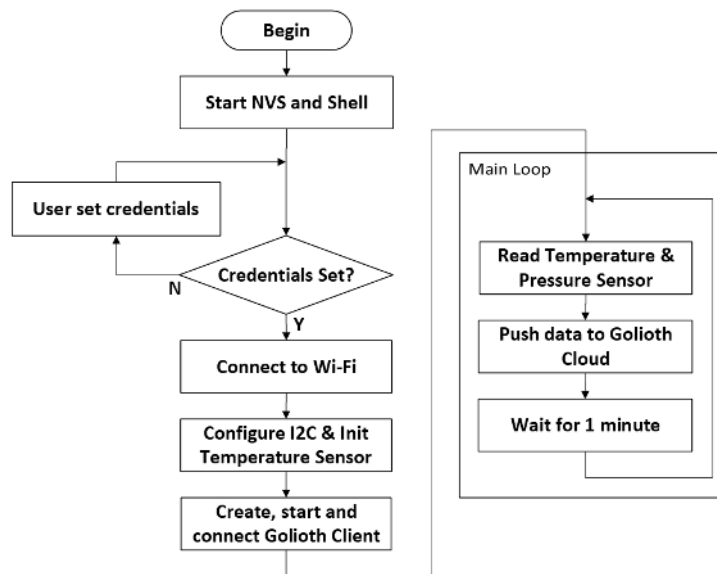


Figure 6.4 – ESP32 based temperature sensor device code flowchart

The components used in the `app_main.c` function developed for this example application according to *Figure 6.4* are explained in *Table 6.1*.

Component	Purpose	Description
NVS (Non Volatile Storage)	Store credentials	Enables reboot-safe remote devices
Wi-Fi	Network connectivity	Blocks execution until connected
BMP280 Driver	Sensor interface	Abstracts low-level I2C access
CBOR (Concise Binary Object Representation) Encoder	Payload encoding	Optimized for low-power devices
Goloth Client	Cloud connection	Secure telemetry and OTA
FreeRTOS Sync	Connection gating	Prevents premature publishing

Table 6.1 – Goloth temperature-based sensor components explained

After implementing the example code and before flashing the code to the ESP32 it is required to configure Goloth Cloud to include the device you want to connect. We will discuss this in the next subsection.

Connect a device to the Goloth cloud

To connect a device to the Goloth cloud, you will have to follow these steps:

1. Create an account on their website <https://goloth.io/> by clicking on Sign in or Sign Up if you already have an account.
2. After logging into your account, click the Connect Device button. It will open a console asking you to create a project that your device will be part of.
3. Create a project named home-assistant-cloud-temperature-logger. The Goloth console main screen after you create your project looks like *Figure 6.5*. Red rectangles are highlighted with the options you will have to use in this section.

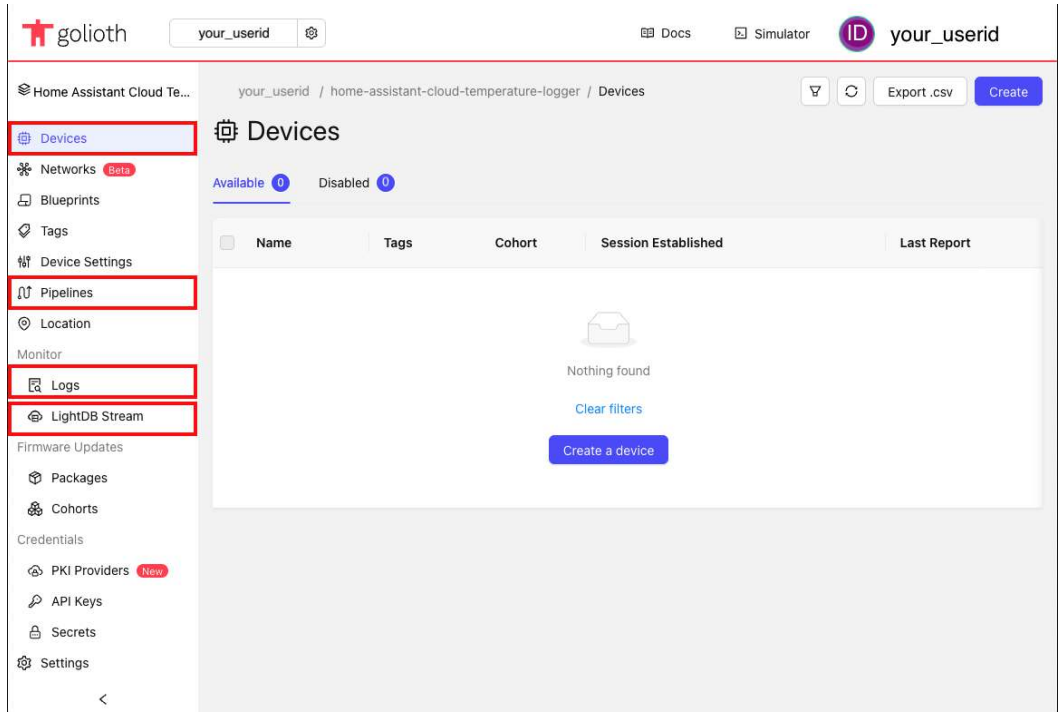


Figure 6.5 – Golioth screen after project creation

- 4. Click on **Devices** in the Golioth console top left to create a Device and then in the create button in the top right of the screen.
- 5. Enter a name for your device and it will be created.
- 6. If you click on the device you just created, and then Credentials, you will be able to see the identity (PSK-ID) and the Pre-Shared Key (PSK). As an example, part of the credentials of my device are shown in Figure 6.6.

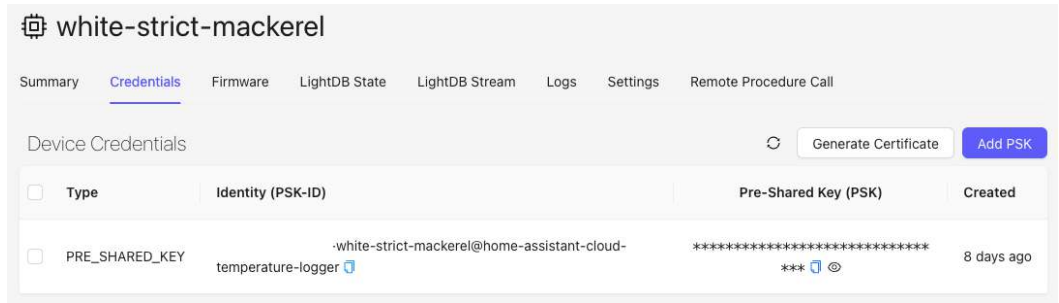


Figure 6.6 – Golioth device credentials

The credentials presented in *Figure 6.6* have to be included after flashing the device with the code presented in the previous subsection. These credentials will allow the device to be connected to Golioth Cloud and this process, as we discussed, is called provisioning. This is the next sequence of steps required to connect the ESP32 device to Golioth as follows:

7. Compile the code from the previous section using the two commands below:

```
idf.py set-target esp32
idf.py build
```

8. Flash the device and include the credentials using the command:

```
idf.py flash monitor
```

9. Press enter and use the command `settings set <key> <key_value>` to enter the Wi-Fi and Golioth credentials.
10. Use the reset command to reset the device. This should be typed in the ESP32 serial console as below.

```
esp32> settings set wifi/ssid your_wifi_ssid
Setting wifi/ssid saved
esp32> settings set wifi/psk your_wifi_password
Setting wifi/psk saved
esp32> settings set golioth/psk-id some_number-white-strict-mackerel@home-
assistant-cloud-temperature-logger
Setting golioth/psk-id saved
esp32> settings set golioth/psk psk_number
Setting golioth/psk saved
esp32> reset
```

If you set up the credentials correctly, your device will be connected to Golioth cloud. If you click on Monitor | Logs in the screenshot presented in *Figure 6.5* you will be able to see some information from the logs. You also see the data from Temperature Sensor by clicking on the LightDB Stream tab in *Figure 6.5*. The data will be presented as in *Figure 6.7*.

LightDB Stream

Time	Device	Data
9:09:09 PM	white-strict-mackerel	⌵ { "pressure_hpa": 931.22314453125, "temperature_c": 24.850000381469727 }
9:08:02 PM	white-strict-mackerel	> {"pressure_hpa":931.1004638671875,"temperature_c":24.81999969482422}
9:07:02 PM	white-strict-mackerel	> {"pressure_hpa":931.1212768554688,"temperature_c":24.84000015258789}
9:06:10 PM	white-strict-mackerel	> {"pressure_hpa":931.0203857421875,"temperature_c":24.84000015258789}
9:05:10 PM	white-strict-mackerel	> {"pressure_hpa":931.0045166015625,"temperature_c":24.790000915527344}
9:04:02 PM	white-strict-mackerel	> {"pressure_hpa":931.0596313476562,"temperature_c":24.809999465942383}
9:03:10 PM	white-strict-mackerel	> {"pressure_hpa":931.0787353515625,"temperature_c":24.790000915527344}
9:02:05 PM	white-strict-mackerel	> {"pressure_hpa":931.0739135742188,"temperature_c":24.690000534057617}
9:01:42 PM	white-strict-mackerel	> {"pressure_hpa":931.03662109375,"temperature_c":24.760000228881836}

Figure 6.7 – Golioth LightDB stream screenshot

Golioth devices always send data to the cloud, but nothing is stored until a Pipeline explicitly routes that data to a destination such as LightDB Stream. In the software code developed, I kept the default CBOR data source path pipeline configured, so you don't need to set up anything else in the Pipeline menu highlighted in *Figure 6.5*.

Now that the ESP32 sensor is connected to the Golioth cloud and streaming temperature data reliably, we have a working remote telemetry pipeline in place. In the next section, you will configure Home Assistant to pull this telemetry data from the Golioth cloud using a REST sensor integration.

Integrating Golioth data with Home Assistant

The idea of using cloud data in Home Assistant is not to replace local automation — since you can still connect to devices directly — but to add capabilities that are otherwise difficult or impossible to manage locally.

Bringing data from Golioth into Home Assistant allows you to maintain a single operational view of your smart home while still benefiting from cloud-native features as discussed.

Before pulling data into Home Assistant, two modifications are needed in the firmware developed in the previous section. The first modification will be to add an option for Golioth to output state data instead of streaming data. The second modification is to adapt the code to receive Over-the-Air (OTA) updates, one of Golioth's strong features.

Configuring the temperature sensor to output Golioth states

The Stream data structure for Golioth allows the sensor information to be stored in the cloud with a timestamp associated. It will consume data storage from the cloud that we don't need for this example developed for the book. Golioth provides a simplified data structure called State. While Stream represents the sensor data over time, State represents the current value of the sensor.

We will have to modify the function `env_publish` to add the Golioth function `golioth_lightdb_set` so we can send data in the state format to the cloud. A compile-time flag `#define COMPILER_STREAM` has been added to the code, allowing you to switch between Stream and State output. If `COMPILER_STREAM` is 0 it compiles to state data and if it is 1 compiles to Stream data. The program logic is still the one presented in *Figure 6.4*.

Since the original Golioth Stream example does not enable the LightDB State service, you must enable it in the ESP-IDF configuration before compiling the modified application. Follow these steps:

1. Run the `menuconfig` command in your project directory

```
idf.py menuconfig
```

2. Choose the option Component Config | Golioth SDK Configuration and select the option Golioth LightDB State service.
3. Compile the code and flash it to the unit.
4. Open the Golioth console and navigate to Devices.
5. Click on your device and then LightDB state. You will see something as presented in *Figure 6.8*.

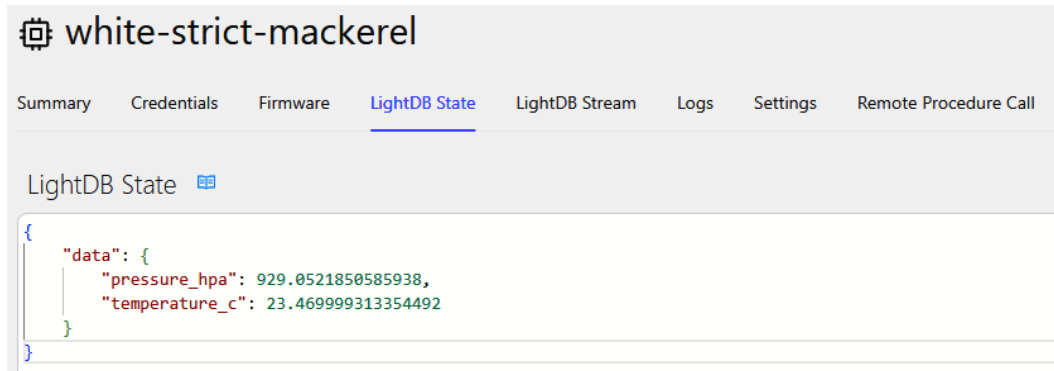


Figure 6.8 – Golioth LightDB state for the temperature sensor

Now that we have our example sending states instead of streams, we will modify the code to add a nice feature implemented in Golioth which is the software update Over The Air (OTA). We will use the next subsection to do that.

Adding OTA feature to the temperature sensor

An important feature provided by Golioth is the ability to update devices Over the Air (OTA). This feature is required for IoT fleet management and is useful when you have to update multiple remote devices in the field where access is difficult or the number of devices is high.

In order to implement this feature, I combined the code developed using the ESP-IDF Golioth stream example as the baseline with the example Golioth fw_update example.

I added the files `fw_update.c` and `fw_update.h` available here: https://github.com/golioth/golioth-firmware-sdk/tree/main/examples/esp_idf/fw_update/main to the main folder of the application we developed. I added the includes `"freertos/FreeRTOS.h"`, `"freertos/semphr.h"` and `"fw_update.h"` to the `app_main.c` file. I also added the variable `*_current_version` to store the firmware version, and called the function `golioth_fw_update_init` before the infinite loop.

A required configuration step is to set up the ESP32 partition for OTA. It was not originally configured, so I had to do so. You can use the `menuconfig` option but to simplify just copy configurations from previous `sdkconfig.defaults` file from the `fw_update` example. The configurations will look like the following:

```
CONFIG_ESPTOOLPY_FLASHSIZE_4MB=y
CONFIG_PARTITION_TABLE_CUSTOM=y
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME="partitions.csv"
```

Create or copy the `partitions.csv` file from the `fw_update` example with the following content:

```
nvs,      data, nvs,      ,      0x4000,  
otadata, data, ota,      ,      0x2000,  
phy_init, data, phy,      ,      0x1000,  
ota_0,    app,  ota_0,    ,      1600K,  
ota_1,    app,  ota_1,    ,      1600K,
```

Note

Changing the partition table modifies the ESP32 flash layout. An incorrect configuration may prevent the application from booting or OTA updates from working. If this happens, verify the partition table and erase the flash before rebuilding and reflashing the firmware.

You will have to erase the ESP32 flash to configure the partitions above. Do this by using the following command:

```
idf.py erase-flash
```

Then use the following commands to build the new code:

```
idf.py fullclean  
idf.py build
```

To test OTA, modify the firmware version define (`FW_VERSION`) to 2.3 and upload the binary to Golioth. Also modify the code to upload the data to Golioth cloud every 5 minutes instead of the 1-minute interval used before. *Video 6.1* (<https://packt.link/03M5Z>) was recorded to show how to do it using the Golioth Console and how the ESP32 monitor shows the OTA in progress.

Note

For this example, the binary filename to be uploaded using the Golioth OTA feature should be `main.bin` as I used the Golioth default package named `main`.

After these final adjustments to the firmware, we will proceed to pull the data from Golioth cloud into Home Assistant, which we'll cover in the next subsection.

Pull data from Golioth cloud to Home Assistant

This section pulls the temperature sensor data from Golioth cloud to Home Assistant using the Representational State Transfer (REST) based sensor method. It consists of implementing a REST sensor (<https://www.home-assistant.io/integrations/sensor.rest/>) that queries the Golioth Application Programming Interface (API) periodically and gets the latest temperature and pressure state information and stores it in the sensor values created in Home Assistant.

As mentioned, Golioth implements an HTTPS API where you can handle data using methods like POST and GET. For this application, we will use the GET method to access the data from the device in our project in Golioth. For more information about Golioth API, please access this link: <https://docs.golioth.io/reference/management-api/openapi>.

So, we will need the following information to get the data using the Golioth API:

- Golioth Project ID: in this case will be home-assistant-cloud-temperature-logger
- Device ID: you will get this information using the following path Devices | Summary | Device ID at Golioth main screen presented in *Figure 6.5*.
- API Key: you will have to create one API Key using the path Credentials | API Keys at Golioth main screen presented in *Figure 6.5*.

After getting all the information above, implement a REST sensor in the Configuration.yaml using the following pseudo code:

```
rest:
  - resource: "https://api.golioth.io/v1/projects/home-assistant-cloud-temperature-logger/devices/<your_device_id>/data"
    headers:
      x-api-key: !secret golioth_api_key
      Accept: "application/json"
    scan_interval: 60
    sensor:
      - name: "Golioth Temperature"
        value_template: "{{ value_json.data.data.temperature_c }}"
        unit_of_measurement: "°C"

      - name: "Golioth Pressure"
        value_template: "{{ value_json.data.data.pressure_hpa }}"
        unit_of_measurement: "hPa"
```

In the case of the example above, the API key credential is stored in the `secrets.yaml` file in Home Assistant. The `scan_interval` is configured for 60 seconds and adjusted according to your convenience. Before restarting Home Assistant, validate the API endpoint using the following curl command:

```
curl -i \
  -H "x-api-key: <your_api_key>" \
  "https://api.golioth.io/v1/projects/<your_project_id>/devices/<your_device_id>/data"
```

Restart Home Assistant to get the sensor values available. Using the REST sensor created above, a Dashboard was created as an example. The data presented in this dashboard is presented in *Figure 6.9*.

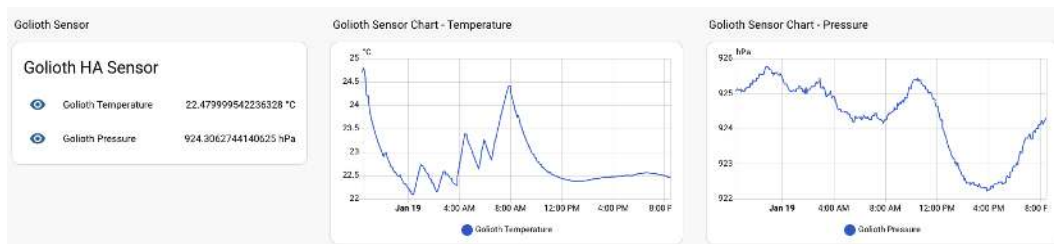


Figure 6.9 – Golioth sensor information presented in Home Assistant

The example presented shows how to bring data from Golioth into Home Assistant. You can also implement the opposite direction by sending data from Home Assistant to Golioth and then to a sensor, relay, or other connected device. This can be accomplished using the HTTP POST method or Golioth's Remote Procedure Call (RPC) feature. Exploring these approaches is left as an exercise for you. For more information and implementation examples, refer to the official Golioth RPC documentation: <https://docs.golioth.io/application-services/rpc/>.

If you follow the sensor example in this and the previous subsection, at this point, your sensor device is connected to Golioth and generating temperature and pressure data. By bridging this data back into Home Assistant using the patterns described above, you complete the loop combining cloud-scale device management with local control and monitoring. In the last subsection from this chapter, you will step back and compare this approach with other cloud-based IoT platforms, such as AWS, Azure and Particle, to understand where Golioth fits and what trade-offs these alternatives introduce.

Exploring other cloud IoT platforms

This section discusses the main IoT cloud platforms available on the market today: AWS IoT Core, Microsoft Azure IoT Hub, Particle Cloud and Arduino Cloud.

AWS IoT Core

AWS IoT Core (<https://aws.amazon.com/iot-core/>) is a cloud IoT platform designed for large-scale and commercial deployments. It is part of the Amazon Web Services ecosystem and is commonly used in industrial, enterprise, and consumer product environments.

AWS IoT Core supports both microcontroller-based devices and Linux-based systems. On the microcontroller side, devices such as ESP32, STM32, and Nordic nRF52 can be used, typically running FreeRTOS with the AWS IoT SDK. For more powerful applications, single-board computers like Raspberry Pi or industrial gateways running Linux are very common.

The primary protocol used to connect devices to AWS IoT Core is MQTT over TLS, secured with X.509 certificates. HTTPS is also supported, mainly for provisioning or firmware downloads. AWS provides OTA updates through AWS IoT Jobs, but the firmware update logic must be fully implemented by the developer.

AWS IoT Core is powerful and flexible, but it introduces significant complexity. It is best suited for large fleets or commercial products rather than small Home Assistant projects.

Microsoft Azure IoT Hub

Microsoft Azure IoT Hub (<https://azure.microsoft.com/en-us/products/iot-hub>) is Microsoft's enterprise IoT platform, designed for industrial, corporate, and regulated environments. It integrates deeply with Azure services such as security, analytics, and digital twins.

Azure IoT Hub supports a wide range of hardware, from microcontrollers like ESP32, STM32, and nRF52, to Linux-based gateways and industrial PCs. Microcontrollers typically run FreeRTOS or bare-metal firmware with the Azure IoT SDK, while larger devices often use Linux.

Devices connect to Azure IoT Hub using MQTT, AMQP, or HTTPS, all secured with TLS. Azure provides OTA functionality through Azure Device Update, which supports staged rollouts and version tracking. Like AWS, OTA requires careful firmware design and additional cloud configuration.

Azure IoT Hub is a strong choice when enterprise identity, security, and lifecycle management are required, but it is often too complex for hobbyist or maker-level smart home projects.

Particle Cloud

Particle (<https://www.particle.io/>) offers a tightly integrated IoT ecosystem that combines hardware, firmware, and cloud services into a single solution. Unlike AWS and Azure, Particle strongly favors simplicity and fast onboarding.

Particle Cloud works only with Particle hardware, such as the Photon, Argon, Boron, and Tracker devices. These boards use Particle-managed microcontrollers (including Nordic and STM32 variants) and run Particle Device OS, which abstracts most low-level firmware details.

Devices connect to the Particle Cloud using CoAP (Constrained Application Protocol) over DTLS (Datagram TLS). OTA updates, logging, and fleet management are built in and require minimal developer effort.

Particle is ideal for rapid prototyping and commercial pilots, but it introduces strong vendor lock-in and limited flexibility compared to open ESP32-based solutions.

Arduino Cloud

Arduino Cloud (<https://cloud.arduino.cc/>) is an IoT platform designed for simplicity and ease of use, especially for beginners, educators, and rapid prototyping.

Arduino Cloud works primarily with Arduino-branded microcontrollers, such as Arduino Uno R4, Nano ESP32, MKR WiFi, and Portenta boards. These devices typically run Arduino's firmware framework rather than a full RTOS, although some advanced boards support FreeRTOS underneath.

Devices connect to Arduino Cloud using MQTT over TLS, managed automatically by Arduino libraries. The platform provides dashboards, basic automation, and simple OTA updates for supported boards. However, the system is intentionally abstracted, limiting low-level control and customization.

Arduino Cloud can integrate with Home Assistant indirectly using MQTT, but it is generally not designed for complex or hybrid local-first architectures. It is well suited to learning IoT concepts, simple dashboards, remote monitoring, and educational or proof-of-concept projects.

For more information about Arduino Cloud, I recommend the book *Arduino IoT Cloud for Developers* by Muhammad Afzal (Packt).

Table 6.2 presents the main IoT cloud platforms compared to Golioth

Feature	Goloth	AWS IoT Core	Azure IoT Hub	Particle Cloud	Arduino Cloud
Target devices	MCUs first	MCU + Linux	MCU + Linux	Particle-only	Arduino boards
Supported microcontrollers	ESP32, STM32, nRF52	ESP32, STM32, nRF52	ESP32, STM32, nRF52	Different MCUs (Particle Boards)	Arduino (Nano ESP32, MKR, Portenta, Uno R4)
Linux devices	Optional	Common	Common	No	No
Main protocols	MQTT, CoAP	MQTT, HTTPS	MQTT, AMQP, HTTPS	CoAP	MQTT
OTA updates	Native, simple	Yes (IoT Jobs)	Yes (Device Update)	Native, built-in	Limited (board-dependent)
Vendor lock-in	Low	Medium	Medium	High	Medium
Home Assistant integration	Easy	Indirect	Indirect	Indirect	Indirect
Best use case	Hybrid smart homes, makers	Enterprise fleets	Industrial IoT	Rapid prototyping	Learning, simple IoT projects

Feature	Golioth	AWS IoT Core	Azure IoT Hub	Particle Cloud	Arduino Cloud
Security	TLS/DTLS, PSK or X.509	TLS, X.509, IAM	TLS, X.509, SAS Tokens	TLS, Device Credentials	TLS, Device Authentication

Table 6.2 – Main IoT cloud platforms comparison

As can be seen in *Table 6.2*, while each cloud IoT platform serves a different purpose, solutions designed with an embedded-first mindset tend to integrate more naturally with local automation systems. Platforms that reduce cloud-side complexity, support a wide range of microcontrollers, and provide built-in device management features make it easier to build hybrid architectures. In this context, Golioth stands out as a practical choice for advanced smart home projects, enabling reliable cloud connectivity while keeping Home Assistant at the center of control.

With an understanding of how different cloud IoT platforms compare and where each one fits best, you can now make informed decisions about when or not to introduce cloud services into a Home Assistant based automation system.

Summary

This chapter explains how cloud IoT platforms can be used alongside Home Assistant, not as a replacement, but as an extension of a local smart home system. It showed situations where cloud connectivity can be used, such as devices installed outside the home network, mobile or roaming devices, long-term data storage, and managing many devices from a single place.

Golioth was introduced as a cloud platform designed for embedded devices like the ESP32. The chapter covered its main components, security model, telemetry handling, configuration features, and support for over-the-air firmware updates. A practical example showed how to connect an ESP32 temperature sensor to Golioth, send data to the cloud, update firmware remotely, and bring that data back into Home Assistant so it can be used in dashboards and automations. The chapter also briefly compared Golioth with other cloud platforms helping readers understand the differences and trade-offs between them.

Now that you have seen how cloud platforms can support remote and large-scale device deployments, the next chapter shifts the focus back to the device itself. You will learn how to

use ESPHome to build custom firmware for ESP-based devices, giving you full control over sensors and actuators while keeping a simple and tight integration with Home Assistant.

Part 3

Custom Device Development with ESPHome and MQTT

In this part, you'll build custom IoT devices using ESPHome and MQTT while working with sensors, GPS modules, displays, and cameras. By the end of this part, you'll be able to develop and integrate your own smart devices with Home Assistant.

This part of the book includes the following chapters:

- *Chapter 7, Creating Custom Devices with ESPHome*
- *Chapter 8, Exploring the MQTT Ecosystem*
- *Chapter 9, Building Touchscreen Display and Camera Devices*

7

Creating Custom Devices with ESPHome

This chapter introduces ESPHome, a powerful framework that enables users to create custom firmware for ESP-based microcontrollers using simple YAML configuration. This firmware integrates seamlessly with Home Assistant. You will begin by understanding the core concepts behind ESPHome, its internal architecture, and the advantages it offers over other firmware solutions.

The chapter then guides you through the complete ESPHome workflow, starting with installing the required tools and setting up your development environment. You will learn how to write ESPHome YAML configurations for common sensors and actuators, how configuration is translated into compiled firmware, and how to upload that firmware to devices using both USB and over-the-air (OTA) updates. We will be using an Off-the-shelf Wi-Fi based Door Sensor as a device example. Special emphasis is placed on ESPHome's integration with Home Assistant, including automatic device discovery, native API-based communication, and entity representation within the Home Assistant interface.

Advanced topics are introduced to push beyond basic configurations, including the use of templates, lambdas, and custom components, as well as implementing complex logic directly on the device to reduce latency and dependency on the central automation system. The chapter also includes a hands-on project in which you build an advanced motion and vibration sensor using a six-axis IMU (Inertial Measurement Unit) and ESPHome, demonstrating a practical application of custom firmware development.

By the end of this chapter, you will be able to develop custom firmware for ESP devices using ESPHome, compile and upload ESPHome firmware, including OTA updates, integrate ESPHome devices seamlessly with Home Assistant, implement on-device logic and

automations using ESPHome, and interface and communicate with a motion and vibration sensor and process its data for home automation.

We will cover the following topics in this chapter:

- Introducing ESPHome
- Configuring, compiling, and uploading an ESPHome device
- Integrating ESPHome devices with Home Assistant using the native API
- Implementing advanced ESPHome features and on-device logic
- Building a Smart Trash Collection Monitor using the MPU-6500

Technical requirements

To get the most out of this chapter, readers should be comfortable working with Home Assistant and understand how devices, entities, and automations interact within the platform. Familiarity with YAML syntax is important, as ESPHome configurations rely heavily on declarative YAML files and templates. From a hardware perspective, basic knowledge of motion and inertial sensors such as how accelerometers detect movement, vibration, and orientation, and how these sensors communicate with microcontrollers over interfaces like I²C will help readers better understand the examples and design choices presented.

The hands-on examples in this chapter require the following hardware and software:

- A Home Assistant installation with the ESPHome Device Builder app installed
- An ESP32, ESP8266, or compatible ESPHome-supported development board
- A Tuya Wi-Fi Door Sensor based on the CB3S (BK7231N) module
- An MPU-6500 6-axis IMU breakout board (or a compatible MPU-6050 module)
- An 18650 battery module and one or two 18650 Li-ion batteries for the Smart Trash Collection Monitor project
- A USB-to-3.3 V serial adapter (FTDI or equivalent) for the initial firmware upload
- The `litchiptool` utility for flashing BK7231-based devices
- Basic electronic prototyping tools, such as jumper wires, a breadboard, and a regulated power supply

All ESPHome YAML configurations, Home Assistant automations, and supporting files presented throughout this chapter are available in the book's GitHub repository.

You can download the code for all examples in this book by following the instructions in the **Download the example code files** section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Introducing ESPHome

ESPHome (<https://esphome.io/>) is an open-source framework designed to simplify the creation of custom firmware for ESP-based microcontrollers used in smart home projects. Instead of writing low-level C or C++ code, developers describe device behavior using a declarative YAML configuration, which ESPHome then converts into fully compiled firmware.

ESPHome was originally created in 2019 by Otto Winter (<https://github.com/ottowinter>) to address the growing complexity of maintaining custom firmware for Home Assistant–connected devices. The project quickly gained adoption within the Home Assistant community due to its tight integration model and focus on local, reliable device control. In 2021, Nabu Casa acquired ESPHome and today it is actively maintained by the Home Assistant project, with development led by the Home Assistant core team and contributions from the open-source community.

ESPHome exists to close the gap between hardware development and home automation integration. It allows engineers and advanced users to focus on what devices should do such as sensing, reporting, and controlling by abstracting firmware, build systems, and communication protocols. This design approach makes ESPHome especially well-suited for rapid prototyping and custom hardware for smart home devices.

Understanding ESPHome's role in Home Assistant systems

ESPHome is tightly integrated with Home Assistant, to the point where it feels like a native extension rather than a third-party tool. Devices flashed with ESPHome are automatically discovered by Home Assistant and exposed as entities with minimal configuration.

This tight coupling enables a local, low-latency, and secure control path. Sensor data, button presses, and actuator commands are exchanged directly over the local network without requiring cloud services or external brokers. *Figure 7.1* shows the architecture diagram involving ESPHome devices, Home Assistant Core and Automations.

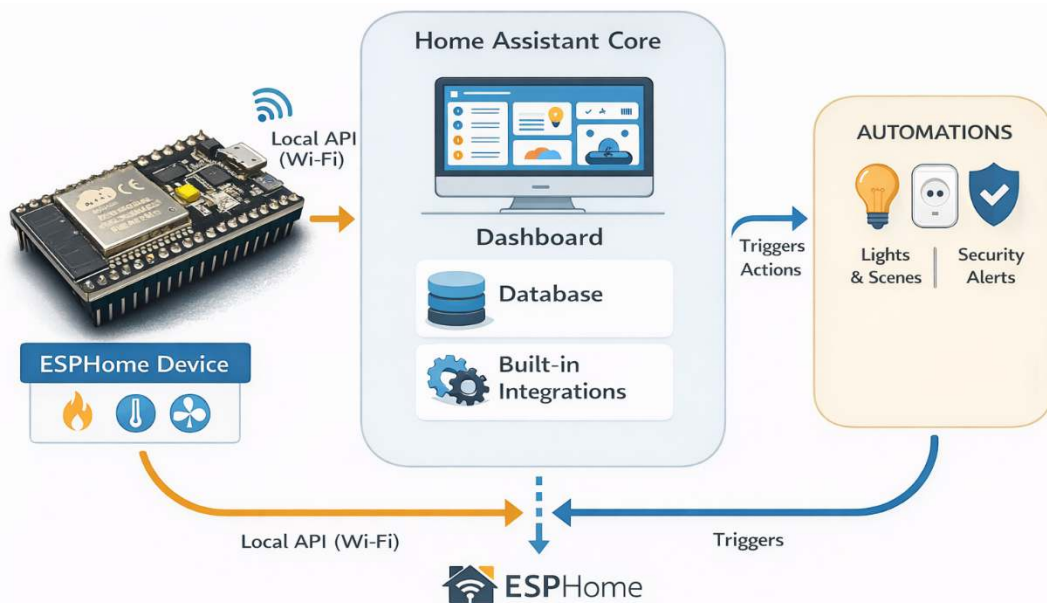


Figure 7.1 – Architecture diagram involving ESPHome and Home Assistant

The diagram in *Figure 7.1* shows an ESPHome device communicating locally with Home Assistant Core, where sensor data is processed and exposed as entities. Home Assistant evaluates this data and triggers automations, which then execute actions such as controlling lights or sending alerts, all within a fast, local control loop.

In Home Assistant deployments, ESPHome often fills the role of:

- A custom device platform for sensors not available off-the-shelf
- A rapid prototyping tool for new automation ideas
- A bridge between custom hardware and high-level automations

Exploring supported hardware platforms, peripherals, and communication buses

ESPHome focuses on Espressif microcontrollers, which offers an excellent balance of cost, performance, and ecosystem maturity. It also supports other SOCs (System On Chip) and microcontrollers.

A complete list of supported microcontroller devices is presented in *Table 7.1*.

Manufacturer	Platform	Variant/Family
Espressif Systems	ESP32	esp32, esp32s2, esp32s3, esp32c2, esp32c3, esp32c5, esp32c6, esp32c61, esp32h2 or esp32p4
Espressif Systems	ESP8266	NA
Raspberry Pi	RP2040	Only Pico W Board including Cypress CYW43439 chip
Beken Corporation	LibreTiny	bk7231t, bk7231n, bk7251, bk7231q
Realtek Semiconductor	LibreTiny	rtl8710b, RTL8720c
Lightning Semiconductor	LibreTiny	ln882hki

Table 7.1 – ESPHome supported microcontroller devices

Table 7.1 summarizes the hardware platforms supported by ESPHome, showing focus on Wi-Fi-enabled SOCs and microcontrollers from multiple vendors. Devices from Espressif Systems form the core of the ecosystem, with broad support for both the ESP32 family including variants such as S2, S3, C-series, H2, and P4 and the widely used ESP8266. In addition, ESPHome supports the RP2040 from Raspberry Pi, specifically the Pico W board, which includes an onboard Wi-Fi chip CYW43439 from Cypress.

Beyond Espressif and Raspberry Pi, ESPHome also extends support through the LibreTiny project (<https://docs.libretiny.eu/>), enabling firmware development for alternative Wi-Fi chipsets from manufacturers such as Beken Corporation, Realtek Semiconductor, and Lightning Semiconductor.

In terms of common peripherals and communication buses, ESPHome abstracts hardware interfaces into reusable components, making it easy to connect external devices without writing driver code. Commonly used peripherals include:

- GPIO for buttons, relays, LEDs, and PIR sensors
- I²C for environmental sensors, ADCs, and expanders
- SPI for displays, flash storage, and high-speed peripherals
- UART for GPS modules, energy meters, and serial sensors
- Deep Sleep mode for battery powered Sensors

Figure 7.2 shows some examples of the common peripherals, sensors, actuators, and communication ports supported by ESPHome. For all available components check this website: <https://esphome.io/components/>

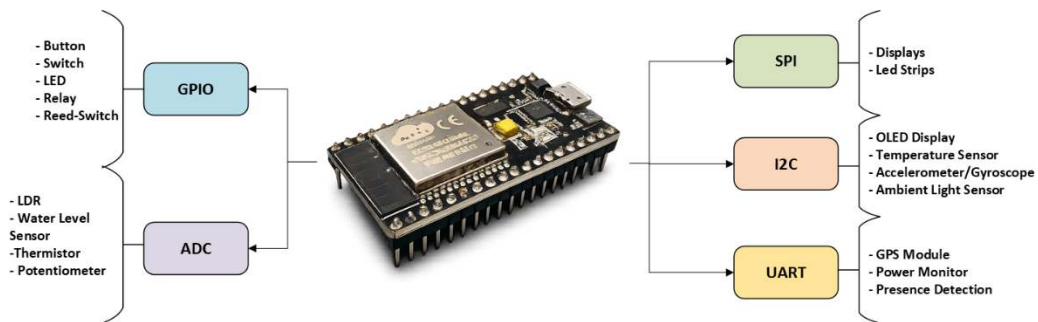


Figure 7.2 – Example of some ESPHome components

Comparing ESPHome with traditional firmware development

Figure 7.3 shows a comparison between traditional embedded systems firmware development versus ESPHome workflow.

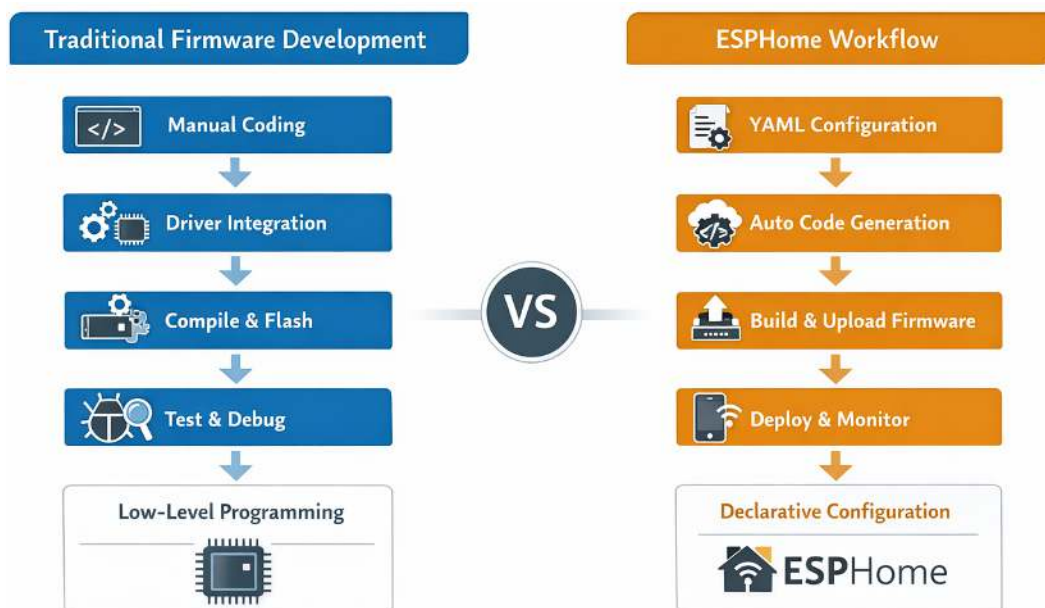


Figure 7.3 – Traditional FW development x ESPHome workflow

Traditional ESP-based firmware development using Arduino or ESP-IDF requires manual coding and project setup, driver integration, protocol implementation, and custom OTA and update logic. ESPHome replaces this with a declarative configuration model, where users describe sensors, outputs, and logic in YAML. From this description, ESPHome generates optimized C++ code and manages the full build pipeline.

Compared to Tasmota, which is another pre-compiled firmware solution, ESPHome is built specifically for tight integration with Home Assistant. ESPHome devices are automatically discovered, communicate through a secure native API, and require minimal manual configuration, resulting in lower latency and a cleaner system architecture without MQTT dependencies used in Tasmota.

Understanding the core concepts behind ESPHome

At its core, ESPHome uses a declarative configuration model. Following the ESP Workflow in *Figure 7.3*, instead of writing code step by step, you describe using YAML syntax what the device has such as sensors, switches, displays and how they should behave. The outcome is a **YAML Configuration** (First block of *Figure 7.3*). This makes parameters easier to read, change, and maintain, even when devices become more complex.

The next step in the ESP Workflow is the **Auto Code Generation** (second block of *Figure 7.3*) where ESPHome automatically turns the YAML configuration into C++ code and compiles it

using the Arduino or ESP-IDF toolchain. The result is a firmware image built specifically for the selected board and hardware, without the user needing to manage build systems or write low-level code.

If you read my first book or are familiar with upload firmware using USB to Serial programmer, the third step in the ESP Workflow is common to the traditional approach. This consists of **Build and upload Firmware** to the device (Third block of *Figure 7.3*). You will have to do the physical step one time per device, since the next uploads you can do using the Over the Air (OTA) feature.

The **Deploy and Monitor** is the last step in the ESP workflow and comes after the firmware upload. Once the device boots with the new firmware, it connects to the network and integrates with Home Assistant. This stage covers runtime behavior: verifying sensor data, checking logs, observing entity states, monitoring connectivity, and performing OTA updates over time.

For communication, ESPHome uses a native encrypted API with Home Assistant by default, offering fast and local control. MQTT is also available when integration with other systems is needed. In both cases, ESPHome devices are fully integrated into Home Assistant, supporting automatic discovery and OTA updates directly from the UI.

With a clear view of how ESPHome connects hardware, interfaces, and components into a single device, the next section moves into the ESPHome workflow itself, showing how these configurations are written, built, flashed, and maintained in practice.

Configuring, compiling, and uploading an ESPHome device

ESPHome follows a workflow that turns YAML files into firmware running on devices introduced in *Table 7.1*. Understanding this workflow is required for building custom devices, troubleshooting issues, and scaling from one prototype to many deployed nodes. Rather than writing embedded code manually, you focus on describing the device, while ESPHome handles code generation, compilation, and communication with Home Assistant.

At a high level, the workflow can be broken into four main phases: configuration, code generation, compilation, and uploading. These phases are the same whether ESPHome is used as a Home Assistant app, via Docker, or through the command-line interface (CLI).

We will use an off-the-shelf Tuya battery-powered door sensor as an example to configure a new device, starting in this section. This sensor is based on the Beken BK7231N chip built into the CB3S Wi-Fi module and is presented disassembled in *Figure 7.4*. We will evolve the code of

the sensor in each section of this chapter as we are presenting the ESPHome features accordingly.

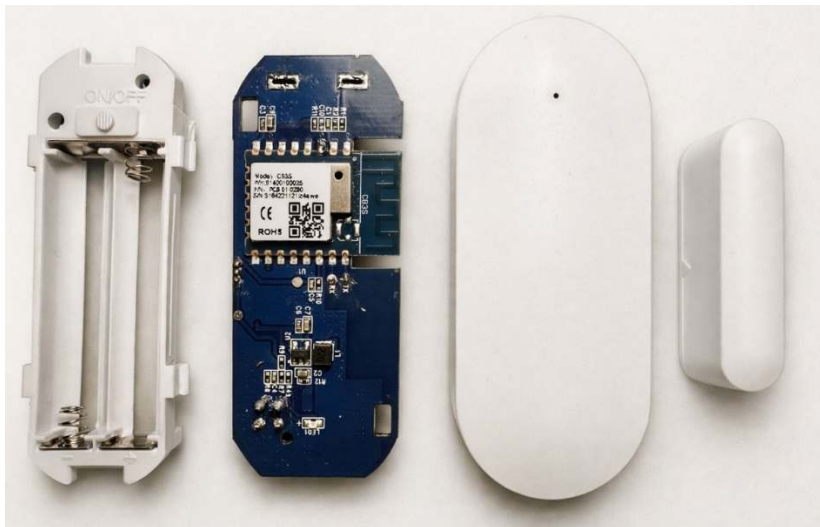


Figure 7.4 – Off-the-shelf battery powered Tuya door sensor used as example

Configuring the ESPHome device

Install ESPHome via Settings | Apps | Install app | ESPHome Device Builder.

After installing ESPHome, you will have to create a new device using the app by clicking the + New Device button. There are three options to create a configuration for a new device: New Device Setup, Import from File and Empty Configuration.

We will use the New Device Setup option to create a device called ESPHome Door Sensor, choosing the BK72xx device, CB3S Wi-Fi Module and installing an Encryption key. The following configuration file will be created:

```
esphome:
  name: esphome-door-sensor
  friendly_name: ESPHome Door Sensor

bk72xx:
  board: cb3s

# Enable Logging
logger:
```

```
# Enable Home Assistant API
api:
  encryption:
    key: "api_key"

ota:
  - platform: esphome
    password: "ota_password"

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password

# Enable fallback hotspot (captive portal) in case wifi connection fails
ap:
  ssid: "Teste Fallback Hotspot"
  password: "ap_password"

captive_portal:
```

The file just created defines the essential building blocks required for the device to connect, communicate, and receive updates. Each section can be described as follows:

- **esphome:** defines the internal device name and a friendly name that appears in Home Assistant.
- **bk72xx:** specifies the hardware platform and board type. In this case, the firmware targets a BK7231-based CB3S module using the LibreTiny framework.
- **logger:** this component enables serial logging for debugging and diagnostics during development and runtime. ESPHome offers 7 log levels as described here: <https://esphome.io/guides/troubleshooting/>
- **api:** this section activates native communication with Home Assistant. The encryption key secures the connection between the device and the Home Assistant server.
- **ota:** this block enables Over-The-Air firmware updates. Once the device is initially flashed via USB, future updates can be performed wirelessly using the defined password.
- **wifi:** this section configures network connectivity using credentials stored in the `secrets.yaml` file.

- `ap`: If the device fails to connect to the configured Wi-Fi network, it creates a fallback access point. This allows recovery and reconfiguration without physical access.
- `captive_portal`: this component enables a simple web-based configuration interface when connected to the fallback access point.

After creating this basic configuration, you can now upload this file manually to the Door Sensor device since it is offline as presented in *Figure 7.5*.

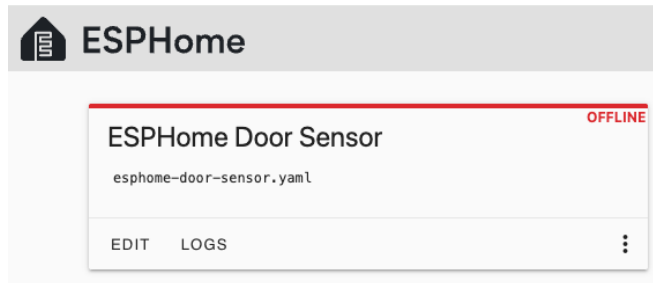


Figure 7.5 – ESPHome door sensor offline

You will have first to create an image file to be manually recorded to the Door Sensor Wi-Fi module. Click on the three dots in the bottom right of *Figure 7.5* and then on Install. Choose the option for Manual Download. The file will start to compile and after all, the message INFO Successfully compiled program will be displayed. Choose the option UF2 package and save the file.

The Door Sensor uses the CB3S Wi-Fi module, and programming it requires a special software tool called *litchiptool* (<https://docs.libretiny.eu/docs/flashing/tools/litchiptool/>). You will have to do this just one time as the subsequent firmware uploads will be done using the ESPHome OTA feature. I created the *Video 7.1* (<https://packt.link/oD37m>) and *Video 7.2* (<https://packt.link/Rx9F1>) to explain how to do it using the *litchiptool* and the wiring configuration to connect the door sensor module board to the computer. You will need an FTDI USB-to-3.3 V serial converter, as shown in *Figure 7.6*, to program the module.

Figure 7.6 shows a simplified wiring diagram illustrating how to connect the FTDI USB-to-Serial converter to the CB3S module. The connections must be soldered directly to the CB3S module or to soldered header pins, as the module does not include a USB programming interface. The battery shown in *Figure 7.6* can be replaced with a regulated power supply between 3.0 VDC and 3.6 VDC.

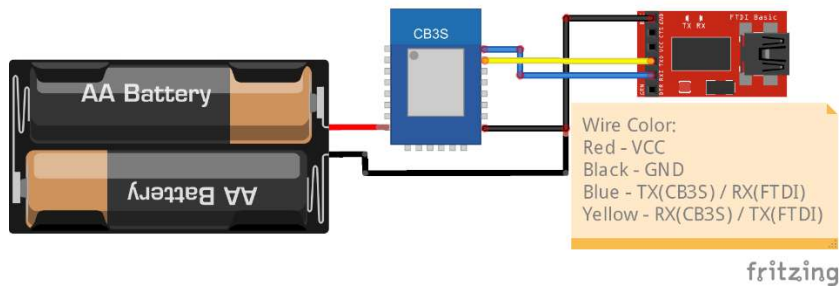


Figure 7.6 – CB3S Wi-Fi module connection to USB to serial converter

Note

The CB3S module operates at 3.3 V. If your FTDI adapter is configured to output 5 V, do not connect it directly to the module, as this may permanently damage the CB3S. Configure the FTDI adapter for 3.3 V operation or use an appropriate logic-level converter when necessary.

After following the sequence of *Video 7.1* (<https://packt.link/T70qM>), we have the device configured and ready to receive updates using OTA. In the next subsection, we will generate some code to be included as part of the device.

Generating code for the ESPHome device

I partially reverse-engineered the Tuya Battery-Powered Door Sensor to identify the connections between the CB3S Wi-Fi module and the surrounding components. The objective was to determine the GPIO connections required to develop the ESPHome firmware. The portion of the circuit relevant to this project is shown in *Figure 7.7*.

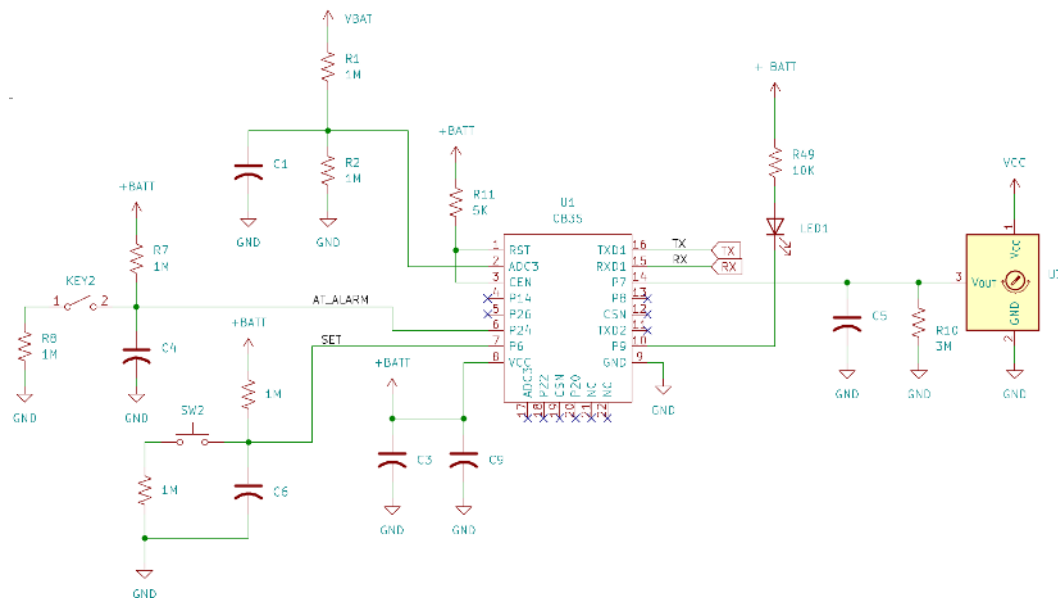


Figure 7.7 – Part of the Tuya door sensor device electronics schematics

From the partial schematic shown in *Figure 7.7* and the CB3S datasheet (<https://solution.tuya.com/hardware/detail/82001>), I identified the connections required to implement the ESPHome firmware. *Table 7.2* maps the CB3S pins to the corresponding signals observed on the door sensor PCB and summarizes how each connection is used in the project.

CB3S Pins	Signal Name	I/O Type	Electrical Behavior	Function in System
P7	Door Sensor (Hall)	Digital Input	LOW = Closed HIGH = Open	Detects door state transitions
P24	Tamper Switch	Digital Input	LOW = OK HIGH = Tamper	Detects enclosure opening or key removal
P6	Setup Button	Digital Input	LOW = Pressed HIGH = Released	Long press enables OTA mode

CB3S Pins	Signal Name	I/O Type	Electrical Behavior	Function in System
P9	Status LED	Digital Output (Active Low)	LOW = LED ON HIGH = LED OFF	Provides visual system feedback
ADC3	Battery Voltage	Analog Input	0–3.3V (scaled via divider)	Measures battery voltage for level estimation

Table 7.2 – Door sensor hardware pin summary

The door sensor is a useful example for introducing basic ESPHome code. We will start by configuring the pins identified in *Table 7.2*. For now, we will use all of them except P6 (Setup Button) and P24 (Tamper Switch), which will be introduced later in this chapter. The ESPHome YAML required to implement the basic functionality is shown below.

```
# Digital Input – Door Sensor (P7)
binary_sensor:
  - platform: gpio
    name: "Door Open"
    device_class: door
    pin:
      number: P7
      mode: INPUT
    on_state:
      then:
        - light.toggle: status_led

# Digital Output – Status LED (Active LOW)
output:
  - platform: gpio
    pin:
      number: P9
      inverted: true
    id: status_led_output

light:
  - platform: binary
    name: "Status LED"
    id: status_led
```

```
    output: status_led_output

# Analog Input - Battery Voltage (ADC3)
sensor:
  - platform: adc
    pin: ADC3
    name: "Battery Voltage"
    update_interval: 30s
    unit_of_measurement: "V"
    accuracy_decimals: 2
    filters:
      - multiply: 2.4  # Adjust based on resistor divider
```

Note

This reverse engineering step is specific to the Tuya device. For development boards and modules with published pinouts, simply use the manufacturer documentation to configure the ESPHome pins.

We will not go into details in the description of the YAML code just presented, but as you can see, the syntax is very intuitive. That is the idea of ESPHome. Simplify the code development so you can concentrate on hardware design, automation logic, and system integration instead of writing C++ from scratch.

Some specific details about some parts of the code are presented in *Table 7.3*.

Code Line	Purpose
light.toggle	Toggles LED on state change
inverted: true	Active LOW output
update_interval: 30s	ADC sampling interval
multiply: 2.4	Voltage divider correction

Table 7.3 – ESPHome YAML syntax specifics

To include the code, you will need to click Edit shown in *Figure 7.5*. The YAML code editor will be shown. After you include your code, click Save.

The next step after including the YAML code is to validate and install it in the device. We will cover this in the next subsection.

Compile and upload the code to the ESPHome device

To compile the code to the ESPHome device one option is from the YAML code editor by clicking on the Install button at the up right corner of the screen. ESPHome will validate the YAML syntax and then compile your code in the same command. I created *Video 7.3* (<https://packt.link/8Kq4N>) which shows how to compile the code.

The code will be validated, compiled and installed in the ESPHome door sensor device using the OTA feature. After the firmware upload using OTA, if everything goes well, the messages "INFO OTA successful " and "INFO Successfully uploaded program" will be shown in the compile/upload/log window. If you go to settings | Devices & services | Devices and look for ESPHome Door Sensor the sensor will be listed there. By clicking on the sensor, you will see the sensor configuration in Home Assistant as presented in *Figure 7.8*.

If the upload or integration does not work as expected, check the following common issues:

- YAML validation or compilation errors
- Incorrect Wi-Fi SSID, password, or network connectivity
- Incorrect OTA password or encryption key
- Device not reachable on the network
- Wrong pin assignments or unsupported hardware configuration
- Home Assistant not discovering the ESPHome device
- API connection or firewall issues between the device and Home Assistant

The ESPHome logs and Home Assistant device logs are the first places to check when troubleshooting these problems.

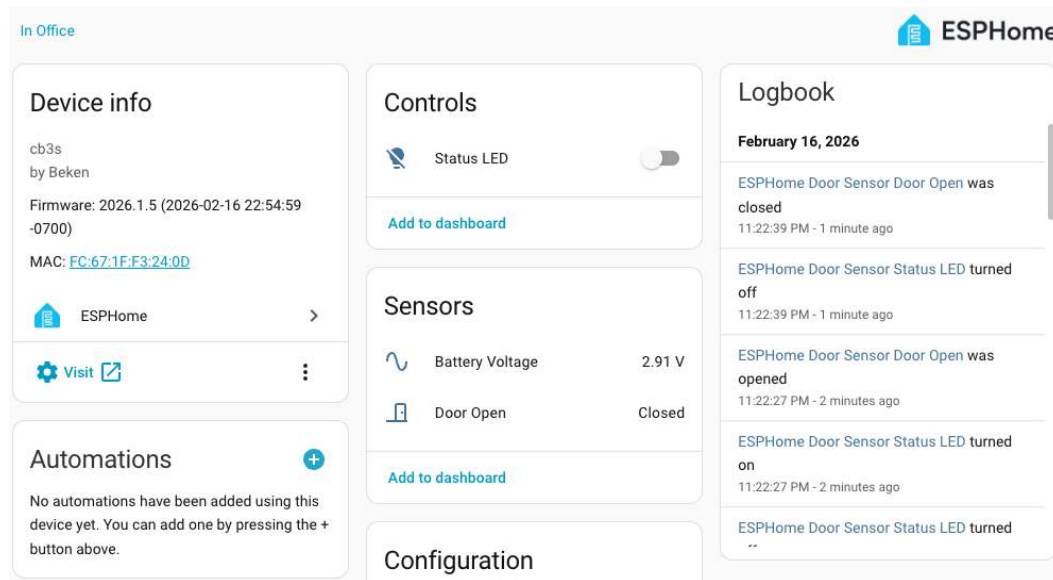


Figure 7.8 – ESPHome door sensor status on Home Assistant

Figure 7.8 also shows some tests done and presented in the Logbook at right with the Door Open sensor changing the status to open and close and the status led following it as expected.

Now that the ESPHome-based door sensor is integrated into Home Assistant and its core functionality is validated, we are ready to move to the next section. We will explore how ESPHome devices communicate with Home Assistant.

Integrating ESPHome devices with Home Assistant using the native API

One of the advantages of ESPHome is its native integration with Home Assistant. ESPHome communicates directly with Home Assistant using its own native API instead of other firmwares like Tasmota that relies on MQTT. This creates a tighter, reliable, and structured integration between the devices and Home Assistant.

In this section, you will learn how ESPHome devices are discovered, how entities are created and registered internally, how devices are organized inside Home Assistant, and how to design clean and maintainable device representations.

Using the native ESPHome API

By default, ESPHome devices use the native ESPHome API for communication. When you include `api:` in your YAML configuration, the device establishes a persistent connection to

Home Assistant. This connection by default is encrypted, and the encryption can be removed if required.

The boot sequence for an ESPHome device is described in *Figure 7.9*.

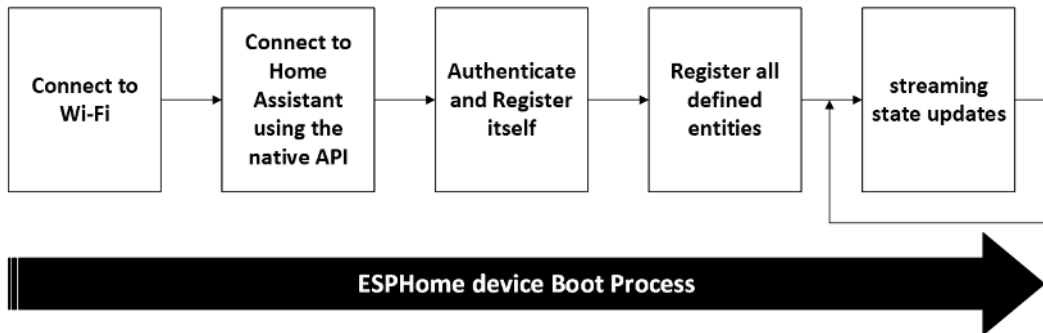


Figure 7.9 – ESPHome device boot process

During and after the boot process presented in *Figure 7.9*, because ESPHome uses a direct native API connection with Home Assistant, it enables automatic device discovery, real-time two-way communication, encrypted data exchange, built-in over-the-air (OTA) firmware updates, and automatic synchronization of entity metadata.

Note

For more information about ESPHome API check the following website: <https://esphome.io/components/api/>

Understanding ESPHome device discovery and entity creation

As presented in the previous section, once flashed and connected to the same network, an ESPHome device is automatically discovered by Home Assistant. No manual YAML configuration is required on the Home Assistant side.

This discovery is possible because ESPHome sends structured information about every entity it defines, including names, unique IDs, device classes, and measurement units. The ESPHome API explicitly registers each entity and links it to a device entry in the Home Assistant Device Registry.

Every component defined in ESPHome YAML becomes an entity inside Home Assistant. For example, if we create the Tamper Sensor for the Door Sensor using the code:

```
binary_sensor:
  - platform: gpio
    name: "Tamper"
    device_class: tamper
    pin:
      number: P24
      mode: INPUT
```

It will be added with the others already created as presented in *Figure 7.10*.

↑	Entity	Device	Area	Integration
^	ESPHome Door Sensor			
🔋	Battery Voltage	ESPHome Door Sensor	Office	ESPHome
🚪	Door Open	ESPHome Door Sensor	Office	ESPHome
💡	Status LED	ESPHome Door Sensor	Office	ESPHome
⚠️	Tamper	ESPHome Door Sensor	Office	ESPHome

Figure 7.10 – Tamper entity added to ESPHome door sensor device

The Tamper sensor will have following configuration in Home Assistant:

```
binary_sensor.esphome_door_sensor_tamper
|
domain          device name          component
```

During the API handshake between ESPHome and Home Assistant, ESPHome sends structured metadata including: Entity name, Unique ID, Device class, Unit of measurement, State class, Icon and Availability status. Home Assistant stores this information in the Device Registry and Entity Registry, which is why entities survive reboots and firmware updates. The system understands that a particular entity belongs to a specific physical device.

ESPHome automatically exposes useful diagnostic entities such as: Wi-Fi signal strength, Uptime, IP address and Firmware version. These entities are categorized as diagnostic sensors. They are valuable in advanced installations because they allow you to monitor network quality, detect weak Wi-Fi areas, identify unstable devices and track firmware consistency across devices.

Now that you understand how ESPHome devices are discovered, registered, and managed inside Home Assistant, the next step is to unlock their advanced features. In the next section, we will move beyond basic integration and explore advanced ESPHome features, including custom components and on-device logic, allowing you to push intelligence closer to the hardware and build more powerful smart devices.

Implementing advanced ESPHome features and on-device logic

As your smart home grows, moving part of your logic from Home Assistant to the edge device becomes increasingly important. In this section, we explore how ESPHome allows you to implement advanced behavior directly on the microcontroller, using templates, lambdas, scripts, globals and deep sleep strategies.

You will learn how to transform Home Automation devices into intelligent edge nodes capable of making local decisions, filtering noisy signals, and optimizing network traffic.

Using lambdas and templates

One of ESPHome's most powerful features is called lambdas. Lambdas are small blocks of inline C++ code embedded in YAML. It is the mechanism that allows custom logic inside ESPHome.

You can use lambdas in:

- Template sensors
- Filters
- on_boot triggers
- State changes (on_press, on_release, on_state)
- Display rendering
- Value transformations
- Deep sleep conditions

A template sensor in ESPHome is different from what we saw in *Chapter 3* about Jinja2 template syntax. In ESPHome, a template is a component type. It creates a virtual sensor whose value is calculated locally on the device.

Using our recently created ESPHome door sensor as an example, we can create a template sensor for the battery sensor that uses a lambda feature to give us the battery percentage value based on the value measured by the battery voltage sensor we added before. We can do that using the YAML code below:

```
- platform: template
  name: "Battery Level"
  id: battery_level
  unit_of_measurement: "%"
  update_interval: never
  lambda: |-
    float v = id(battery_voltage).state;
    if (v >= 3.3) return 100;
    if (v <= 2.8) return 0;
    return (v - 2.8) * 100.0 / (3.3 - 2.8);
```

This template sensor computes the battery level percentage locally on the BK7231N device using an inline C++ lambda block. The `lambda: |-` line is particularly important: the `|` symbol in YAML introduces a multi-line literal block, allowing you to write properly formatted C++ code across several lines, while the `-` suppresses the final newline. Inside this block, C++ logic is executed at runtime. The code reads the voltage from `battery_voltage`, stores it in a `float` variable, and then applies conditional statements to determine the battery percentage. If the voltage is at or above 3.3V, it returns 100%; if it is at or below 2.8V, it returns 0%; otherwise, it performs a linear interpolation to calculate a proportional value between 0 and 100.

Implementing on-device automation with actions and triggers

ESPHome provides a set of actions that define what the device should actually do when an automation runs. Actions are the execution layer of on-device logic and can include tasks such as turning GPIO outputs on or off, publishing MQTT messages, calling Home Assistant services, introducing delays, running scripts, setting global variables, logging messages, or putting the device into deep sleep. Actions can also be combined with `if`, `while`, and `repeat` constructs to build structured logic directly on the microcontroller. By chaining multiple actions together, you can create fully autonomous behaviors without relying on Home Assistant to coordinate each step. It's important to mention that these actions allow your device to keep running logic even if Home Assistant or the Wi-Fi network is offline.

In addition to Actions, ESPHome implements several built-in Triggers that allow you to execute actions directly on the device when specific events occur. The `on_press` trigger runs when a binary sensor changes from OFF to ON, making it ideal for detecting button presses or door openings. On the other hand, `on_release` executes when the sensor transitions from ON

to OFF, which is useful for detecting when a button is released or a door is closed. The `on_value` trigger activates whenever a sensor's value changes, enabling reactive behavior for analog or numeric inputs like temperature or voltage. Finally, the `interval` trigger runs actions periodically at a defined time interval, making it suitable for scheduled tasks such as periodic measurements or status updates.

We can add the `on_state` trigger to the following code to flash the `status_led` for 500ms when the `door_open` sensor changes state:

```
on_state:
  then:
    - light.turn_on: status_led
    - delay: 500ms
    - light.turn_off: status_led
```

The `on_boot` trigger runs automatically when the device starts up, allowing you to initialize variables, restore states, or perform startup logic. We will use this trigger to add some features to our door sensor later in this section.

For more information about actions and triggers check here: <https://esphome.io/automations/actions/>.

Creating reusable scripts and global variables

The `script:` block encapsulates reusable sequences of actions that can be called from multiple triggers without duplicating logic. This promotes clean design and improves readability. For example, the code presented in the previous subsection to flash the `status_led` can be included in a script and reused in multiple locations:

```
script:
  - id: slow_single_flash
    mode: parallel
    then:
      - light.turn_on: status_led
      - delay: 500ms
      - light.turn_off: status_led
```

You can simply include `script.execute: slow_single_flash` in your code to execute the script.

The `globals:` section is used to store runtime variables that persist while the device is powered, and optionally across reboots if `restore_value` is enabled. These variables can be accessed and modified anywhere in the configuration using `id()`, making them ideal for tracking state information such as flags, timestamps, counters, or operating modes. As an example, we can implement the following code:

```
globals:
  - id: ota_mode
    type: bool
    restore_value: false
    initial_value: "false"
```

We will use this global in our final application to allow the door sensor to be on to get updates via OTA.

Optimizing power consumption with Deep Sleep

The ESPHome door sensor implementation has a hard constraint which is battery power consumption. The Tuya CB3S datasheet says that when the module is connected to Wi-Fi, the average power consumption is 25mA. That said, if the module is just connected to Wi-Fi without any communication, it will drain the two batteries connected to the sensor in less than two days. Fortunately, ESPHome implements the Deep Sleep mode. Deep Sleep mode puts the microcontroller device to sleep so the battery consumption drops to the microamperes range extending the battery charge duration.

In our final door sensor implementation, I included all the advanced features that we saw before combined with the Deep Sleep mode. The final code implements the state chart diagram presented in *Figure 7.11*.

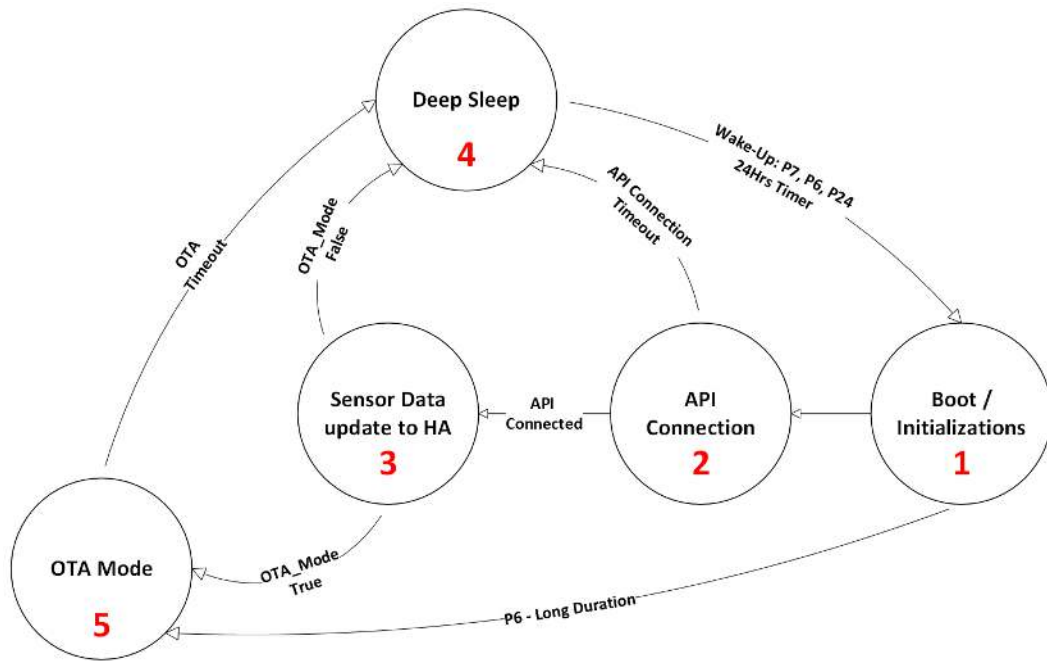


Figure 7.11 – ESPHome door sensor state diagram architecture

The possible state sequences presented on Figure 7.11 are:

- 1 → 2 → 3 → 4: After Boot (1), if the device establishes an API connection within the 60-second timeout (2) it publishes sensor data to Home Assistant (3), and then enters deep sleep (4) provided OTA mode is not active.
- 1 → 2 → 4: After boot (1), if the device fails to establish an API connection before the timeout expires (2), it skips sensor updates and transitions directly to deep sleep (4).
- 1 → 5 → 4: After boot (1), if the setup button (P6) is detected as pressed during initialization, the device enters OTA mode (5), remains awake for updates, and transitions to deep sleep (4) only after the OTA timeout expires.

It is important to emphasize that after the initial boot cycle completes and the device transitions to Deep Sleep, this becomes its default operational state to preserve battery life; from that point forward, the unit will only wake under four specific conditions: a door state change on pin P7 (GPIO wake interrupt), a setup button press on P6, a tamper input trigger on P24, or the expiration of the 24-hour deep sleep timer, after which the full boot sequence (state 1) executes again.

The deep sleep sequence is configured using the directive `deep_sleep`: and became available for the BK7231 chip in the version available at the time I'm writing this chapter (2026.1.5). That is maybe why this feature is not completely mature for the application implementation yet. During the tests, a state transition for the `door_open` sensor took about 18 seconds to be updated to Home Assistant which is a lot compared to the original Tuya application.

In regards to battery duration, as an example, assume the door is opened and closed nine times per day (18 wake-up events in total) and the device remains in the disabled state (approximately 330 μ A per CB3S datasheet) between events. Using a 1,000 mAh battery, the theoretical battery life can be approximated as:

Battery life $\approx 1,000 \text{ mAh} / 0.33 \text{ mA} \approx 3,030 \text{ hours} \approx 126 \text{ days}$ (about 4 months).

This simplified calculation assumes that the current consumed during the short wake-up periods is negligible compared to the time spent in the disabled state. In practice, battery life will depend on the number of door events, the Wi-Fi connection time, battery characteristics, and the current consumed by the remaining circuitry.

Now that we have explored the advanced features of ESPHome let's move to an advanced sensor integration. In the next section, we will build a Smart Trash Collection Monitor Sensor, demonstrating how ESPHome can implement a real-world custom application device.

Building a Smart Trash Collection Monitor using the MPU-6500

This project was born from a very practical and familiar problem in my house. Every Wednesday, before 8:00 AM, the recyclable trash must be placed outside for collection. Despite having a structured automation system at home, my wife and I would occasionally forget to move the bin to the street. The result was simple but frustrating: an entire extra week of accumulated recyclable waste. Rather than relying on memory or manual reminders, this solution was designed to automatically verify whether the trash bin was placed outside on time and confirm that it was actually collected.

Summarizing what we discussed in the last paragraph, the goal of this system is to:

1. Detect when the trash can is moved toward the street.
2. Alert us if it has not been placed outside before Wednesday 8:00 AM.
3. Detect the collection event (tilt + vibration pattern).
4. Confirm successful pickup.
5. Reset automatically for the next week.

This combines motion sensing, time-based logic, and pattern recognition into a single IoT device.

The Smart Trash Collection Monitor is divided into three layers:

1. Hardware layer
2. ESPHome firmware
3. Home Assistant automation logic

We will explain each layer in the next subsections.

Hardware layer

For this implementation we will use the following hardware components to build our prototype solution:

- ESP32 Wi-Fi Module
- MPU-6500 accelerometer/gyroscope module
- 18650 Battery Module
- Sealed 3D printed enclosure

The ESP32 was chosen due to its Wi-Fi stability and more processing power. The MPU-6500 sensor is obsolete but the breakout module board is still found in many retailers. ESPHome does not natively support MPU-6500, but register compatibility allows reuse of MPU-6050 platform which is another accelerometer. It provides 3-axis acceleration which will be used to calculate vibration for the rolling movement when taken outside and the tilt angle or inversion when the trash can is emptied. It can also be used to detect the vibration impact when the trash can is dropped back down.

The 18650 battery module provides either 3 or 5Vdc. The module I used supports two 18650 batteries in parallel. Each battery can provide 3500mAh so 7000mAh in total using two batteries. Doing a rough estimate in the worst case, this can lead to more than one month of battery duration assuming the module will be in deep sleep and waking up just when the trash can is moved.

Because the device will be mounted in an outdoor trash can, a proper enclosure is necessary to protect the electronics against weather exposure, including rain, condensation, humidity, dust, and temperature variations.

The interconnection diagram for the electronic modules is presented in *Figure 7.12*.

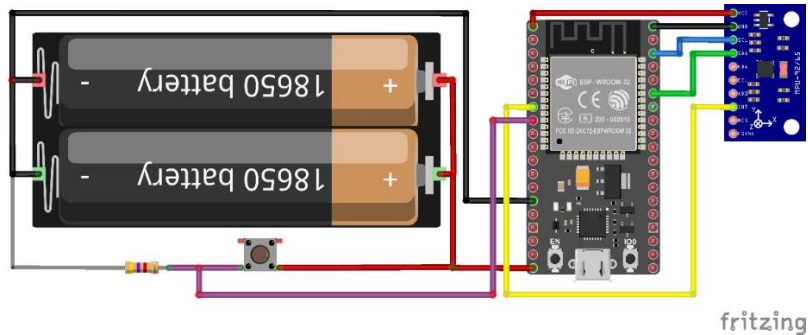


Figure 7.12 – SmartTrash electronics prototype interconnection

The MPU-6500 sensor communicates to the ESP32 over I2C, so the SDA and SCL pins in the MPU-6500 breakout board are connected to the ESP32 GPIO 21 and GPIO 22 respectively. The ESP32 GPIO33 pin is connected to the MPU-6500 INT pin which is an Interrupt to wake up the ESP32 when a movement is detected.

And the GPIO32 is connected to a push button to enable the OTA mode when pressed.

The 3.3Vdc line in the ESP32 module supplies the voltage to the MPU-6500 board. The 18650 battery module supplies 5Vdc to the ESP32 module.

ESPHome firmware

The MPU-6500 ESPHome I2C implementation is done using the `i2c:` component. You have to declare the ESP32 pins that will be connected to it. The code to implement the I2C bus is the following:

```
i2c:
  id: i2c_bus
  sda: 21
  scl: 22
```

As of ESPHome 2026.1.5, the MPU-6500 doesn't have its own platform under the sensor component, but it shares most of its configuration registers with the MPU-6050, which has an ESPHome component available. This makes the MPU-6500 ESPHome code development easier. The values for the acceleration are obtained by declaring `accel_x`, `accel_y`, `accel_z` variables. We will need just acceleration values so the YAML code to make them available is:

```
sensor:
  - platform: mpu6050
    id: mpu
```

```

address: 0x68

accel_x:
  id: accel_x
accel_y:
  id: accel_y
accel_z:
  id: accel_z

```

The reason we are using the MPU-6500 unit instead of the MPU-6050 is because the 6500 has a Wake-on-Motion interrupt. This interrupt works by changing the INT pin level when a sample from any axis exceeds a user-programmable threshold. This interrupt will be useful when we implement the deep sleep, which we will explain in the next paragraph. ESP32 needs to send a sequence of configuration commands via I2C to set up the Wake-on-Motion interrupt. This sequence of steps is presented in *Table 7.4*. More details about the Wake-on-Motion interrupt configuration can be found in the MPU-6500 Product Specification datasheet.

Step	Register Name	Address (Hex)	Data (Hex)	Purpose
1	PWR_MGMT_1	0x6B	0x00	Wake device (SLEEP=0, CYCLE=0)
2	PWR_MGMT_2	0x6C	0x07	Enable accel, disable gyro
3	ACCEL_CONFIG2	0x1D	0x01	Set accelerometer Low Pass Filter to 184 Hz
4	INT_ENABLE	0x38	0x40	Enable Motion Interrupt (MOT_INT_EN)
5	MOT_DETECT_CTRL	0x69	0xC0	Enable hardware motion detection (ACCEL_INTEL_EN + ACCEL_INTEL_MODE)
6	WOM_THR	0x1F	0x0F*	Set motion threshold (~60 mg per LSB × value)
7	LP_ACCEL_ODR	0x1E	0x07	Set low-power accel sampling rate (example: 31.25 Hz)

Step	Register Name	Address (Hex)	Data (Hex)	Purpose
8	PWR_MGMT_1	0x6B	0x20	Enable CYCLE mode (Low Power Accel Mode)

Table 7.4 – MPU-6500 Wake-On-Motion (WOM) configuration table

The ESPHome firmware must use the deep sleep feature since the device will be using battery and needs to be attached to the trash can. Without the Deep Sleep, the battery will drain quickly, and the solution will be impractical to be implemented. The firmware wakes up from Deep Sleep mode when a motion is detected, and an interrupt pin changes its state. The device gets the acceleration values and calculates vibration magnitude and computes tilt angle. Depending on the values calculated for vibration and tilt angle, the firmware can trigger events to detect rolling movement for trash taken outside and Trash collected.

The MPU-6500 accelerometer measures acceleration along three axes: X, Y, and Z. Each axis represents one component of the acceleration vector. When the trash can is stationary, the dominant acceleration acting on the sensor is gravity. When the trash can moves, shakes, rolls, or is lifted by the collection truck, these acceleration values change.

To detect high vibration, we combine the acceleration across the three axes by calculating the 3D vector length. This is also known as acceleration magnitude which is calculated by the following formula:

$$\text{Magnitude} = \sqrt{x^2 + y^2 + z^2}$$

The code which implements Magnitude is:

```
- lambda: |-
    float x = id(accel_x).state;
    float y = id(accel_y).state;
    float z = id(accel_z).state;
    float magnitude = sqrt(x*x + y*y + z*z);
```

The tilt angle represents how far the trash can has rotated away from its upright position. In this example, only the acceleration measured along the Z-axis is used. As the trash can tilts, the Z-axis component of gravity changes. Assuming ideal calibration, no sensor bias, and no scale error, the tilt angle can be estimated using:

$$\text{Tilt Angle (degrees)} = \cos^{-1}\left(\frac{z}{9.81}\right) \times \frac{180}{\pi}$$

Note

This calculation assumes the MPU-6500 is mounted with its Z-axis aligned with the vertical direction when the trash can is in its normal upright position. If the sensor is installed with a different orientation, the corresponding axis or a combination of axes must be used, and the tilt calculation should be adjusted accordingly.

The ESPHome code to implement the formula for Tilt angle is the following:

```
float z = id(accel_z).state;
float tilt = acos(z / 9.81) * 180.0 / 3.14159;
```

In this code, `id(accel_z).state` reads the current acceleration along the Z-axis. Dividing this value by 9.81 normalizes it relative to gravitational acceleration. The `acos()` function then calculates the angle in radians, and multiplying by `180.0 / 3.14159` converts the result to degrees. A value close to 0° represents the reference upright orientation, while larger angles indicate that the trash can has rotated farther away from that position.

The magnitude and tilt thresholds must be determined experimentally for the specific installation. In my tests, a magnitude threshold of 5 and a tilt angle above 120° were sufficient to identify meaningful movement and collection events. These conditions are then used to create two binary sensors:

- **Trash Outside:** Detect if the trash can is outside to be collected. It is active within the time frame configured to the trash to be collected or every Wednesday from 8:00 to 10:00AM and if a high vibration is detected.
- **Trash Collected:** Detect if the trash can was collected. It is active within the time frame configured to the trash to be collected or every Wednesday from 8:00 to 10:00AM and if a high vibration is detected and if the trash can is tilted (tilt angle above 120 degrees).

All the logic for weekdays, hours and minutes is obtained from the `time:` component. For more information about the time component, please check here: <https://esphome.io/components/time/>.

Figure 7.13 summarizes the idea behind the firmware implemented using the MPU-6500 IMU module in the Smart Trash Can project.

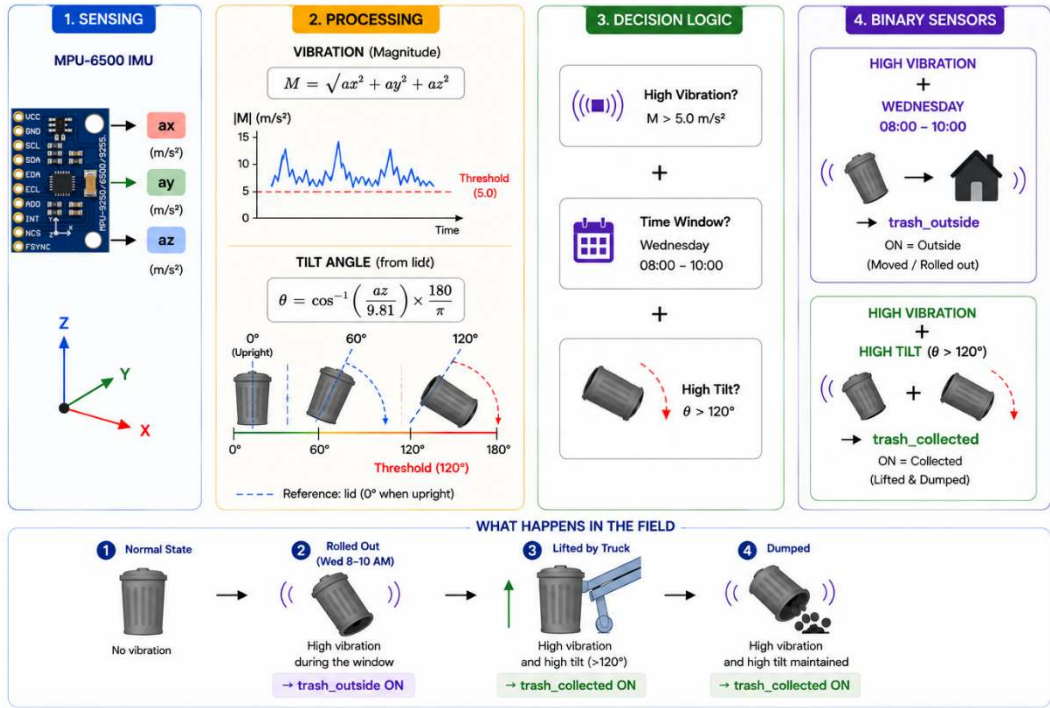


Figure 7.13 – Detecting tilt and vibration using the MPU-6500 IMU

This diagram in *Figure 7.13* illustrates how the Smart Trash Collection Monitor processes data from the MPU-6500 IMU to detect collection events. The ESPHome firmware calculates both the acceleration magnitude and the tilt angle from the sensor readings. High vibration detected during the scheduled collection window activates the `trash_outside` binary sensor, while the combination of high vibration and a tilt angle greater than 120° activates the `trash_collected` binary sensor, allowing Home Assistant to identify when the trash bin has been collected.

The code logic state diagram is presented in *Figure 7.14*.

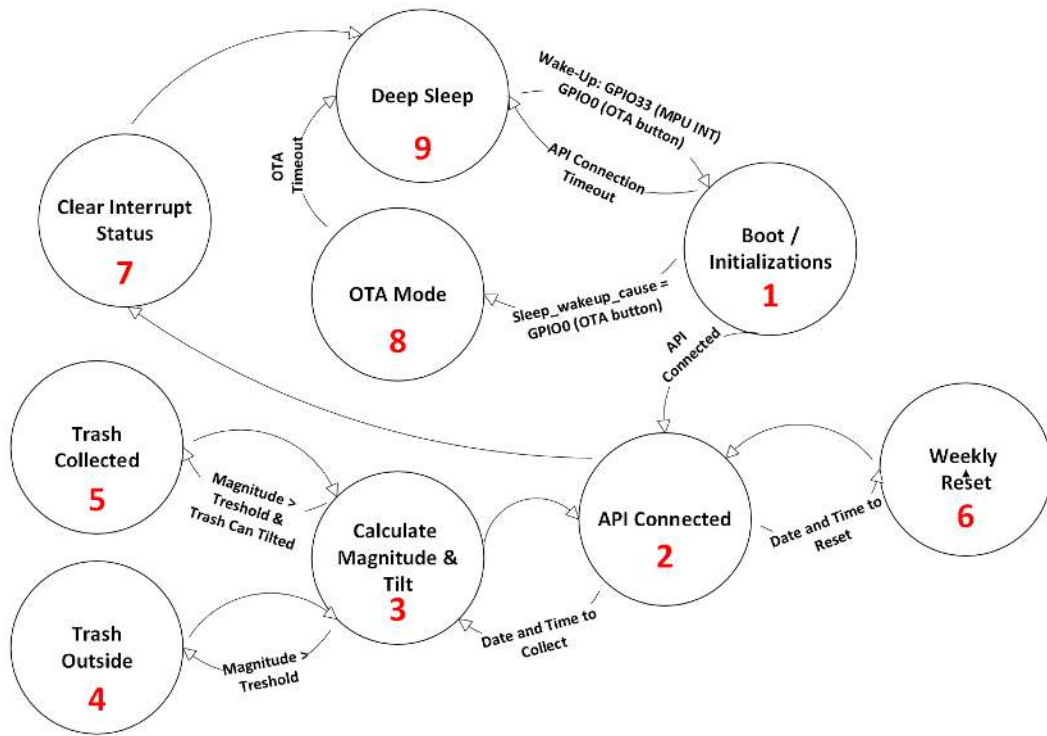


Figure 7.14 – SmartTrash collection monitor state diagram

The normal execution flow starts when the trash can is moved outside. When this happens, the Wake-on-Motion interrupt is triggered, and the device wakes up from Deep Sleep activating the sequence of states 1-->2-->3-->4-->7-->9. What is expected next is that the trash is collected. In this case, the sequence 1-->2-->3-->5-->7-->9 is executed. After that, the trash can is returned to its original position inside the home. When this happens, the sequence 1-->2-->6-->7-->9 will be executed. If the sensor needs a software update, then the OTA switch must be pressed, and the sensor will enter the OTA mode for two minutes and then return to Deep Sleep. In this case, the sequence 1-->8-->9 will be executed.

Home Assistant automation logic

To complete the SmartTrash collection monitor solution, two automations must be implemented in Home Assistant. The first automation is responsible for monitoring if the trash has been placed outside before the scheduled collection time. On the configured collection day, Home Assistant checks the `trash_outside` entity at 7:45 AM. If the sensor has not been triggered, a notification is sent to my mobile device, reminding me to place the trash outside.

This ensures that a missed placement is detected without requiring the battery-powered device to remain awake to track time.

The second automation handles confirmation of collection. Once the accelerometer logic identifies the tipping and movement pattern associated with garbage pickup and sets up the `trash_collected` state, Home Assistant can trigger a notification indicating that the trash has been successfully collected. By delegating scheduling and decision-making to Home Assistant, the ESP32 firmware remains energy efficient. After this notification is received, you can put your empty trash can back to your home.

More solution features can be implemented in the Home Assistant side, like weekday and time frames for collection and trash missed reminder configurations using Home Assistant helpers. I'll leave them to you so you can practice your skills on Home Assistant and ESPHome integrations.

As future implementation plans, my intention is to invest some time in creating the 3D printed enclosure for the solution and also testing the battery charger shield module.

The SmartTrash project demonstrates how ESPHome can transform real-world needs into practical solutions. You are not just limited to off-the-shelf products; you can design, build, and integrate your own intelligent devices.

With these concepts established, we now can conclude the chapter with a summary of its main technical takeaways.

Summary

Chapter 7 introduced ESPHome as a practical framework for creating custom firmware for ESP-based and other devices using simple YAML configuration files. You saw how device behavior can be described clearly in configuration blocks, and how ESPHome generates the firmware automatically. The chapter also explained how ESPHome devices connect directly to Home Assistant through the native encrypted API, enabling automatic discovery and fast local communication.

You followed the complete ESPHome workflow, from writing the configuration to compiling, uploading with OTA, and monitoring the device in Home Assistant. Hardware interfaces such as GPIO, I²C, and ADC were configured easily, and advanced features like lambdas, templates, triggers, and globals showed how logic can run directly on the microcontroller.

The Door Sensor and SmartTrash collector monitor projects demonstrated the ESPHome concepts in real applications. You implemented features in these solutions that would require lots of code and time to be developed if ESPHome was not used. These projects showed how

firmware design and Home Assistant automations work together to create smart home solutions.

More importantly, this chapter provides the foundation for building custom smart devices tailored to your own requirements. ESPHome hides much of the complexity of embedded firmware development, allowing you to implement advanced features through a simple, user-friendly YAML configuration instead of working directly with programming languages, development environments, and compiler toolchains. Combined with Home Assistant, it enables you to create reliable, locally controlled devices that integrate seamlessly into your smart home.

In *Chapter 8, Exploring the MQTT Ecosystem*, we move to MQTT, a lightweight messaging protocol widely used in IoT systems. MQTT introduces a communication model that supports scalable and distributed architectures, expanding your ability to connect devices and systems beyond a single integration model.

8

Exploring the MQTT Ecosystem

Developing custom IoT devices for Home Assistant requires a communication protocol to exchange data between sensors, microcontrollers, and the automation platform. MQTT (Message Queuing Telemetry Transport) is widely used in custom device development because it is lightweight, flexible, and supported by Home Assistant.

This chapter introduces the MQTT ecosystem by covering the fundamentals of the MQTT protocol, the installation and configuration of a Mosquitto broker, and the use of MQTT Explorer and MQTTeX to develop, test, and debug MQTT communication. It concludes with a hands-on project that integrates a GPS module with a microcontroller to transmit location data to Home Assistant, demonstrating how MQTT enables custom devices to support practical automations such as location-based control.

By the end of this chapter, you will be able to understand and apply the MQTT protocol structure and features, set up and secure a Mosquitto MQTT broker for Home Assistant, use MQTT Explorer and MQTTeX to monitor, debug, and test MQTT communication, design effective MQTT topic hierarchies and message payloads, and interface with a GPS module and publish location data through MQTT.

These are the topics covered in this chapter:

- MQTT Fundamentals: Protocol, Broker, Clients, Topics, and QoS
- Setting up and securing the Mosquitto MQTT broker
- Mastering MQTT Explorer for real-time monitoring and debugging
- Advanced MQTTeX usage: testing and simulation
- Hands-on project: GPS tracking and location-based automations

Technical requirements

To follow the examples and exercises in this chapter, you should have:

- A working Home Assistant installation with basic familiarity with its interface and configuration.
- A computer capable of running MQTT Explorer and MQTTeX.
- A basic understanding of MQTT and IoT communication concepts.
- Basic experience with ESP32 or ESP8266 microcontroller development.
- For the hands-on project:
 - A GPS module compatible with an ESP32 or ESP8266.
 - A configured Mosquitto MQTT broker integrated with Home Assistant.
- Basic knowledge of GPS operation to better understand positioning and diagnostic data.

MQTT Fundamentals: Protocol, Broker, Clients, Topics, and QoS

The Message Queuing Telemetry Transport (MQTT) is one of the most adopted protocols on the Internet of Things ecosystem. Designed to operate in constrained environments, MQTT provides a lightweight messaging mechanism that allows devices with limited computing resources and unreliable network connections to exchange data reliably.

In smart home systems, MQTT plays a central role as a communication backbone between sensors, microcontrollers, home automation servers, and user interfaces. Devices such as Wi-Fi Module boards, Zigbee devices, and cloud services can exchange messages through a centralized broker, allowing the entire home automation infrastructure to remain modular and scalable.

MQTT relies on a publish–subscribe architecture as a communication model. This model eliminates the need for devices to communicate directly with each other. Instead, all communication flows through an intermediary server called the MQTT broker.

Because of this architecture, devices can publish information without knowing which systems will consume the data. At the same time, other devices can subscribe to specific categories of messages and automatically receive updates whenever new information becomes available. Let's dive into the MQTT Communication model to understand how this happens.

Understanding the MQTT communication model

In a publish–subscribe system, communication is organized around topics rather than direct device addresses.

When a device generates information, it publishes a message to a specific topic. The MQTT broker receives the message and forwards it to any client that has subscribed to that topic.

This architecture introduces several important benefits for smart home and IoT systems:

- Loose coupling between devices – publishers and subscribers do not need to know about each other
- Efficient bandwidth usage – messages are distributed only to interested subscribers
- Scalability – new devices can join the system without modifying existing components
- Asynchronous communication – devices can operate independently and exchange information when available

In the Home Automation context, a sensor node may publish environmental data while Home Assistant subscribes to the same topic to process automation logic. At the same time, a monitoring application or mobile dashboard could subscribe to the same topic to visualize the information.

In the example from the last chapter about the SmartTrash Collection Monitor, if we adapt the code to communicate via MQTT then we can have the diagram presented in *Figure 8.1*.

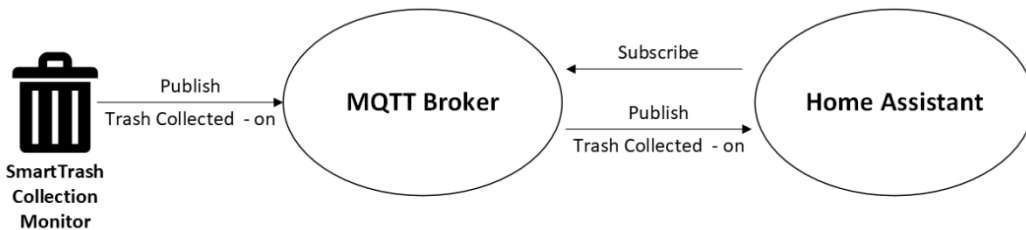


Figure 8.1 – MQTT publish-subscribe example

In *Figure 8.1* we see different MQTT components which we will comment on in the next subsection.

Core MQTT components

The MQTT protocol is built around a small number of core components that define how information flows through the system. The MQTT core components can be listed as following:

- Broker: Central server responsible for receiving and distributing messages
- Client: Device or application that publishes or subscribes to messages

- Topic: Hierarchical string used to organize communication channels
- Payload: The actual data transmitted in the message
- QoS: Quality of Service level defining message delivery guarantees

The broker acts as the central hub of the system. Every client connects to the broker and exchanges messages through it. Once connected, a client can publish data, subscribe to topics, or perform both functions simultaneously.

In a Home Assistant environment, the broker is typically implemented using Mosquitto, while clients may include ESPHome devices, Zigbee gateways, mobile applications, and automation engines.

Topics and topic hierarchies

MQTT messages are organized using topics, which are hierarchical strings that function similarly to file system paths.

Topics allow information to be grouped logically so that subscribers can filter the data they receive.

Examples of typical MQTT topics I used in my home automation applications are:

```
Tele/Garage_Temp_PIR/SENSOR
Tele/ESP32_TEMPERATURE_HUB/SENSOR
cmdn/GarageLights/POWER
```

In this structure:

- Tele represents Telemetry data
- Garage_Temp_PIR, GarageLights represent device names associated with a location and device type
- SENSOR, POWER represent whether it is a sensor or an actuator

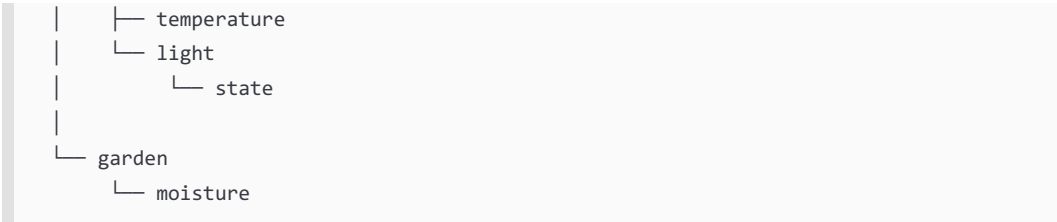
Another option is to use the approach root system – location – sensor data or commands.

Examples of this approach are:

```
home/livingroom/temperature
home/livingroom/light/state
home/garden/moisture
```

The breakdown of this approach is the following:

```
home
├─ livingroom
```



This hierarchical structure enables a flexible and scalable organization of device communications.

MQTT Wildcards

MQTT supports wildcard characters that allow clients to subscribe to multiple topics using pattern matching. This capability is particularly useful in large installations where subscribing to every individual topic would be impractical.

Table 8.1 shows the wildcards available on MQTT.

Wildcard	Function
+	Matches a single level in the topic hierarchy
#	Matches all remaining levels

Table 8.1 – MQTT wildcards

As examples of how to use these wildcards, you can do the following subscriptions:

```
home/+/temperature
```

This subscription receives temperature messages from all rooms.

You can also do the following:

```
home/#
```

This subscription receives all messages within the home hierarchy

Wildcards are frequently used by monitoring tools, logging systems, and debugging applications to capture large volumes of system data.

MQTT Quality of Service (QoS)

Another key concept in MQTT is Quality of Service (QoS), which determines how reliably messages are delivered between clients and the broker.

MQTT defines three QoS levels:

- QoS 0 – At Most Once delivery
- QoS 1 – At Least Once delivery
- QoS 2 – Exactly Once delivery

Table 8.2 summarizes the comparison between the QoS levels.

QoS	Reliability	Overhead	Use Case
0	Low	Low	Sensors
1	Medium	Medium	Status updates
2	High	High	Critical events

Table 8.2 – QoS levels comparison

In this section, we explored the fundamental elements of MQTT, including brokers, clients, topics, and Quality of Service levels. These concepts form the foundation required to build reliable messaging infrastructures that support complex smart home automation systems.

In the next section, we will move from theory to practice by installing and securing an MQTT broker using Mosquitto, which will serve as the communication hub for the projects developed throughout this chapter.

Setting up and securing the Mosquitto MQTT broker

One of the most widely used MQTT brokers in the home automation ecosystem is Mosquitto, an open-source implementation developed by the Eclipse Foundation. Mosquitto is lightweight, highly reliable, and supports all MQTT protocol features including authentication, persistence, and encrypted communication. Because of its simplicity and strong community support, Mosquitto has become the default MQTT broker used in many Home Assistant deployments. It is part of the official Home Assistant apps in the apps store.

We will go over some steps for the Mosquitto installation and configuration in the next subsections.

Mosquitto installation and basic configuration using the Home Assistant app

The app installation process is straightforward and can be completed directly through the Home Assistant interface. Here is a short sequence for installation and basic configuration, assuming you don't have it yet:

1. Open Settings in Home Assistant
2. Navigate to Apps → Install App button
3. In the Search field look for Mosquitto broker and click on icon shown in *Figure 8.2*

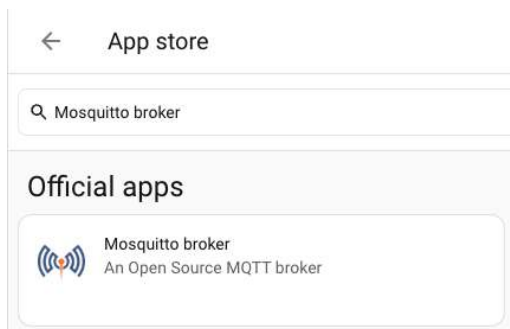


Figure 8.2 – Mosquitto broker app installation icon

4. Click on the Install button as presented on *Figure 8.3*. Mosquitto broker will be installed.

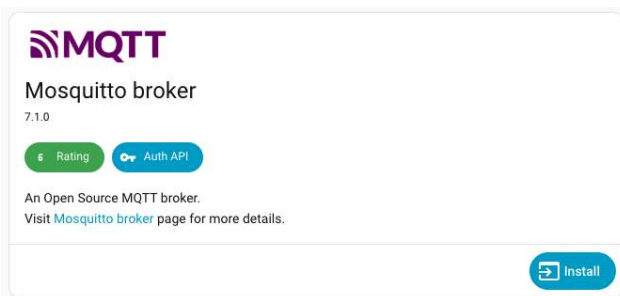


Figure 8.3 – Mosquitto broker installation screen

5. After the installation, the screen on *Figure 8.4* will be presented. Go to Configuration Tab and Configure user credentials.

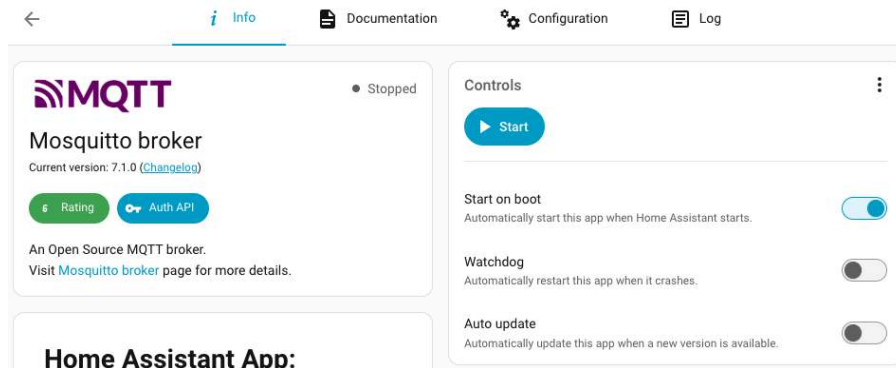


Figure 8.4 – Mosquitto broker initial screen

6. Start the broker service by clicking on the Start button presented in *Figure 8.4*.

Once the broker is running, MQTT clients can connect using the configured credentials to publish messages or subscribe to topics. In the following sections, we will use MQTT Explorer and MQTTX to visualize, publish, and debug MQTT messages in real time.

Mosquitto MQTT authentication and security

Authentication is one of the first security layers that should be enabled in an MQTT deployment. Without authentication, any device on the network could publish or subscribe to topics, which may expose sensitive data or allow unauthorized control of devices.

Mosquitto supports user authentication through configuration. In this case, each user is defined with a username and a hashed password. Devices connecting to the broker must provide valid credentials before they can exchange messages.

MQTT communication can be encrypted using Transport Security Layer (TLS) certificates. Encryption prevents attackers from reading or modifying messages in transit.

In this section, we installed the Mosquitto broker using the Home Assistant app store, reviewed the key configuration parameters, and explored the most important security practices for MQTT deployments.

In the next section, we will explore MQTT Explorer, a powerful tool used to monitor topics, inspect payloads, and debug MQTT communication in real time.

Mastering MQTT Explorer for real-time monitoring and debugging

When developing custom Smart Home devices and automation workflows, visibility into communication between system components is important. MQTT operates as a message-based protocol where devices exchange information through topics and payloads. Without a proper inspection and debug tool, identifying communication problems can become difficult and time consuming.

Among the available MQTT debugging tools, MQTT Explorer is one of the most practical for developers and system integrators. It was created by Thomas Nordquist and provides a graphical interface that simplifies interaction with MQTT brokers and enables immediate visibility of the entire topic structure.

Table 8.3 shows some debug scenarios, examples and how MQTT Explorer can be used to help in the development of Smart Home devices.

Scenario	Example Issue	How MQTT Explorer Helps
Device not appearing in Home Assistant	Wrong topic name	Inspect topic hierarchy
Sensor values incorrect	Payload format mismatch	View raw payload
Automation not triggering	Message not published	Monitor live traffic
Device testing without hardware	Need simulated data	Publish test messages

Table 8.3 – MQTT debugging scenarios x how MQTT Explorer helps

We will see in the next subsections an overview and practical usage of MQTT Explorer main features.

MQTT Explorer overview

MQTT Explorer can be installed using the website: <https://mqtt-explorer.com/>. There are Windows, MAC and Linux versions available for download. After download and installation, you will be asked to configure an MQTT connection. Include the parameters from your Mosquitto MQTT server. As an example, the configurations I'm using in my Home Assistant installation are presented on *Figure 8.5*.

The screenshot displays the MQTT Explorer interface. On the left, a sidebar titled 'Connections' shows a list of connections: 'Mosquitto - Home As...' with URL 'mqtt://homeassistant.local:1883/' and 'test.mosquitto.org' with URL 'mqtt://test.mosquitto.org:1883/'. The main area is titled 'MQTT Connection' and shows the URL 'mqtt://homeassistant.local:1883/'. It includes a 'Name' field with 'Mosquitto - Home Assistant', a 'Validate certificate' toggle (checked), and an 'Encryption (tls)' toggle (unchecked). Below these are fields for 'Protocol' (mqtt://), 'Host' (homeassistant.local), and 'Port' (1883). There are also fields for 'Username' (your_name) and 'Password' (masked with dots). At the bottom, there are buttons for 'DELETE', 'ADVANCED', 'SAVE', and 'CONNECT'.

Figure 8.5 – MQTT Explorer connection screen

If you configure the parameters like Host, Port, Username and Password and just click on the CONNECT button, by default, you will see all MQTT messages exchanged in your Mosquitto server as MQTT Explorer automatically subscribes to all topics using the wildcard #. If you have MQTT devices connected to your server like me, you will see something similar to *Figure 8.6*.

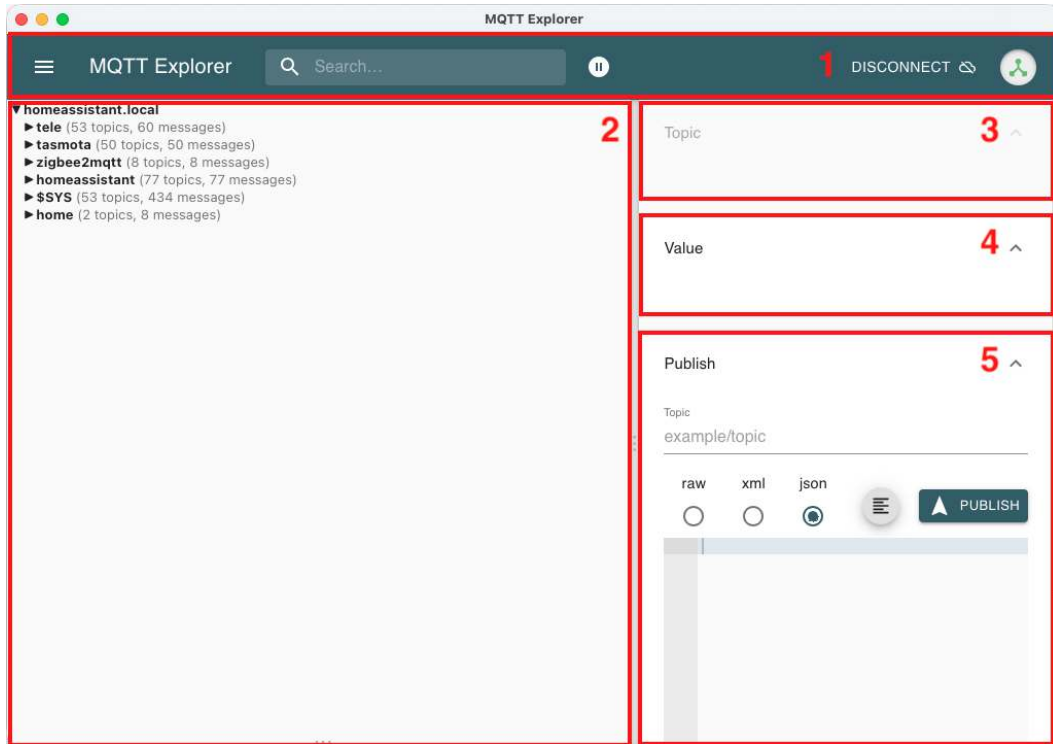


Figure 8.6 – MQTT Explorer main interface overview

Let's describe *Figure 8.6* according to the numbered red rectangles:

1. **Connection and Control Bar:** Displays the current broker connection status and provides controls such as disconnecting from the broker, pausing message updates, accessing settings, and searching across MQTT topics.
2. **Topic Hierarchy Panel:** Shows the complete MQTT topic tree discovered from the broker. Topics are organized hierarchically, allowing users to expand branches and quickly inspect device communication.
3. **Topic Information Panel:** Displays details about the currently selected topic, including its full topic path and metadata such as retained status or message statistics.
4. **Message Payload Viewer:** Shows the most recent payload received for the selected topic and the history of previously received messages during the current session. This panel

allows developers to inspect the message content, verify the format of data being transmitted and observe how values change over time.

5. **Message Publishing Panel:** Allows you to manually publish messages to any MQTT topic. This feature is commonly used for testing devices, simulating events, or triggering automations during development and debugging.

In the next subsections we will learn about the two main important MQTT Explorer usages in my opinion: Message inspection and Message publishing. I created the *Video 8.1* (<https://packt.link/FNSYW>) that shows the examples discussed in both sections.

Inspecting payloads and message metadata

MQTT Explorer not only displays topics. When a topic is selected, it shows the most recent message payload along with useful information that help understand how the system is operating.

You can check if a message is retained, observe how often updates are published, and examine the structure of the payload. When messages use JSON format, MQTT Explorer automatically organizes the data to make it easier to read, which helps simplify debugging messages that contain multiple fields.

Note

A retained message is stored by the broker and automatically sent to new subscribers when they connect to a topic.

As an example, let's evaluate the messages created by the Double Measurement Sensor that combines temperature, humidity, and Motion Sensor status. Instead of clicking in the connect button on *Figure 8.5*, we will click on the Advanced button and filter the messages adding the topic `tele/Garage_Temp_PIR/#` according to *Figure 8.7*. Clicking on the Back button and then on Connect back to the screen on *Figure 8.5*, we will see just the topic we want to verify.

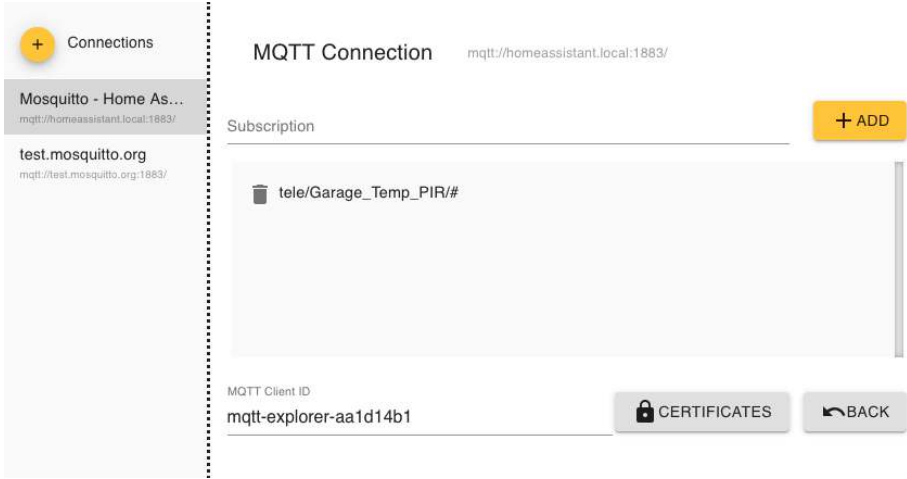


Figure 8.7 – Filter topic to be inspected

We will click on the topic `tele/Garage_Temp_PIR/SENSOR` in the Topic Hierarchy Panel (Figure 8.6 - Red Rectangle number 2) and after some messages being set we will be able to inspect the content of the message sent by this sensor. This message example is shown in Figure 8.8.

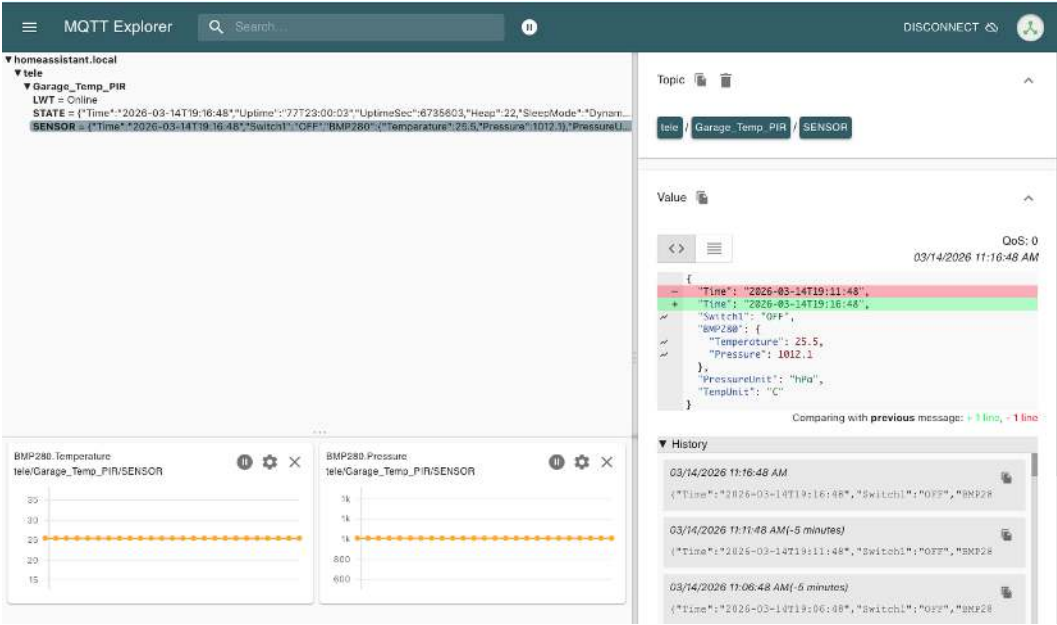


Figure 8.8 – MQTT Explorer message topic inspection

The message inspection panel in the right of *Figure 8.8* allows you to examine the full content of a message published to the topic `tele/Garage_Temp_PIR/SENSOR`. In the example shown in *Figure 8.8*, the payload is formatted as JSON and includes fields such as the timestamp, Motion Sensor switch state, and readings from the BMP280 device, including temperature and pressure values. The interface displays the topic path, the message payload, the Quality of Service (QoS) level, and the time the message was received.

MQTT Explorer also highlights differences between the current message and the previous one, making it easier to identify which values have changed. In addition, the history section shows recently received messages, allowing developers to observe how sensor readings evolve over time. This capability is useful when verifying that devices are publishing data at the expected intervals and with the correct format.

At the bottom of *Figure 8.8*, MQTT Explorer provides user selected trend charts that show how message values change over time. In this example, the charts display the temperature and pressure readings from the BMP280 sensor, updating automatically as new MQTT messages arrive. These charts give a quick visual view of device behavior and help confirm that sensors are publishing data regularly and within expected ranges.

In the next subsection, we will explore how to publish MQTT messages using MQTT Explorer and how this capability can be used to test and debug smart home device development.

Publishing test messages

Another powerful feature of MQTT Explorer is the ability to publish messages manually. This allows developers to simulate device events and test automation logic without needing physical hardware.

I have an automation involving the Double Measurement Sensor and a smart switch installed in my Garage Lights. This automation was created in *Chapter 5* of my first book. It is a convenient light that turns on each time that detects a movement in the garage at night. It turns off the garage lights after 2 minutes. We will use this automation to publish a test message in MQTT Explorer. The idea is to simulate a motion detected in the garage and check if the Smart switch will be activated.

For example, if I publish the test message payload `ON` to the topic `sensor/Garage_Temp_PIR/PIR1`, matching the command expected by the Tasmota device, I can turn on the Garage Lights. *Figure 8.9* shows this test message.

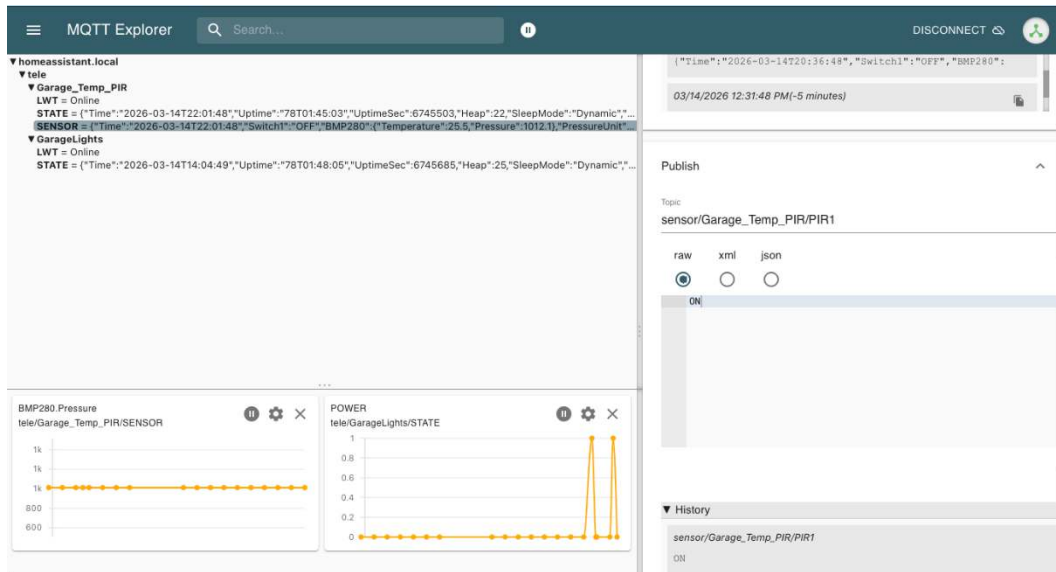


Figure 8.9 – MQTT Explorer test message example

As can be seen on *Figure 8.9* in the Message Publish Panel, the test payload message ON was published in the topic `sensor/Garage_Temp_PIR/PIR1` and the topic `tele/GarageLights/State` changing the parameter POWER from OFF to ON and after 2 minutes turn from ON to OFF. The behavior of the POWER parameter can be seen in the trend chart at the bottom of *Figure 8.9*.

This test message publishing feature is very useful when validating new automations or troubleshooting MQTT related integrations.

MQTT Explorer is an excellent tool for visually inspecting MQTT topics and monitoring message activity. In many development scenarios, however, users require more advanced capabilities to test device communication and simulate MQTT clients. MQTTX provides a powerful environment for publishing, subscribing, and generating MQTT traffic in a controlled way. In the next section, we will learn how MQTTX can be used to test MQTT communication flows and validate the behavior of smart home devices and automations.

Advanced MQTTX usage: testing and simulation

MQTTX focuses more on actively interacting with the MQTT broker, making it an excellent tool for testing devices, validating message structures, and simulating IoT communication during development. It supports multiple simultaneous connections, making it possible to simulate several devices communicating with the same broker.

MQTTX was created and maintained by the company EMQ Technologies Inc. and can be used in different versions like Desktop, Command Line Interface (CLI) or web edition. It is also available for different operating systems such as macOS, Windows and Linux. It is available at <https://mqttx.app/>.

I will provide an overview of MQTTX and then move on to create an example that will be useful to demonstrate its capabilities and to be used in the next section when we develop a GPS positioning sensor to be installed in my car and interact with my garage using an automation.

MQTTX overview

The MQTTX main interface looks like Figure 8.10 after the connection to an MQTT server.

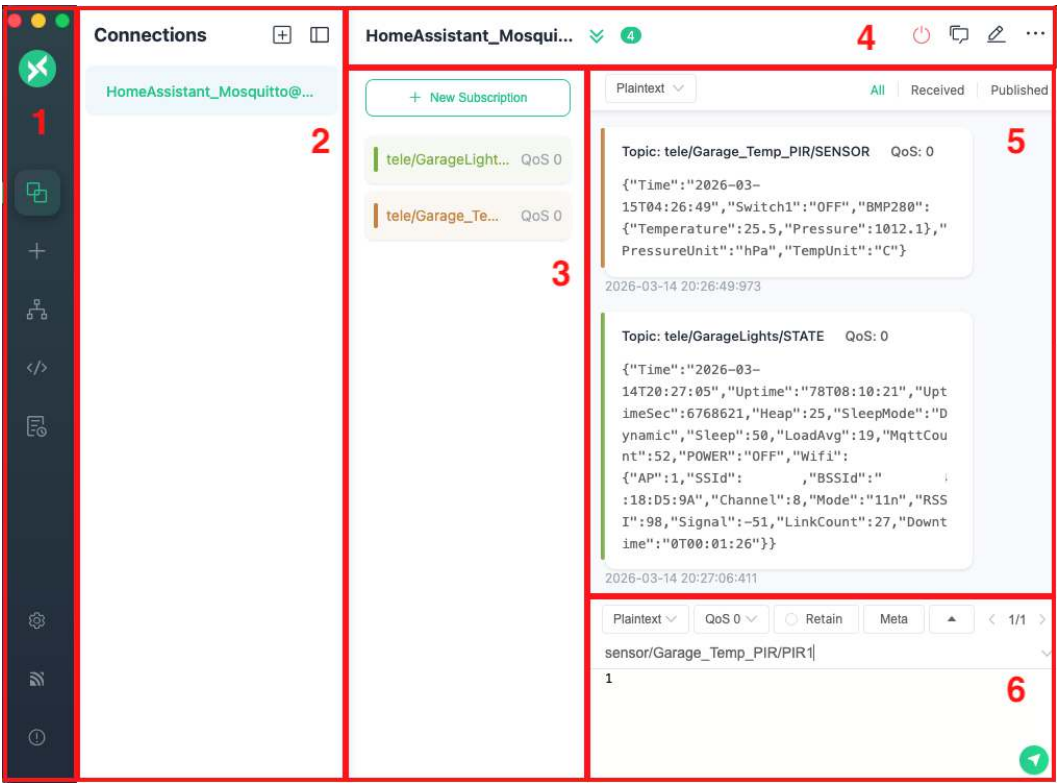


Figure 8.10 – MQTTX main screen overview

Here is a description for each red rectangle area represented in *Figure 8.10*:

1. **Navigation Sidebar:** Provides access to the main MQTTX features, including connections, scripts, message history, and application settings. It acts as the primary navigation menu for the tool.
2. **Broker Connections Panel:** Lists all configured MQTT broker connections. Selecting a connection establishes communication with the MQTT server and loads its associated subscriptions.
3. **Topic Subscriptions List:** Displays the topics currently subscribed to by the client. You can add new subscriptions and monitor the QoS level associated with each topic.
4. **Connection Status and Session Controls:** Shows the active MQTT connection and its status. It also provides controls to disconnect, reconnect, or modify the connection configuration.
5. **Message Monitor:** Displays messages received and published from subscribed topics. Each entry shows the topic, QoS level, timestamp, and the formatted payload content.
6. **Message Publisher Panel:** Used to publish messages to MQTT topics. You can define the topic, payload, QoS, and retained flag to simulate device communication or trigger automations.

Now that we know how the MQTTX interface works, we will use it to build a home automation device the other way around — creating the virtual sensor first, before the physical hardware. We will first create a virtual version of the MQTT ESP32 GPS sensor before building the hardware device. This allows us to simulate data and validate the MQTT topics, Home Assistant entities, and automations reducing debugging time during hardware integration. Let's do this in the next subsections.

Creating a car GPS virtual device using MQTT discovery

In this section, we define a complete MQTT-based GPS virtual device that represents a vehicle equipped with an ESP32 and a GPS module. To do this, we will use MQTT Discovery which is a feature used in Home Assistant that allows Smart Home devices to automatically announce themselves and their capabilities to the MQTT broker, eliminating the need for manual configuration. MQTT Discovery will be used to automatically register a single GPS device with multiple entities, including latitude, longitude, speed, and garage approach detection.

To create the virtual GPS sensor, we will have to publish the JSON payload message below to the topic: `homeassistant/device/car1_gps/config`.

MQTT messages can contain data in different formats, such as plain text, numbers, or JSON (JavaScript Object Notation). JSON is one of the most common formats because it organizes related information into key-value pairs, making it easy for applications such as Home Assistant to extract individual values. In this example, the JSON payload defines the device information and the entities that Home Assistant will create automatically through MQTT Discovery.

```
{
  "dev": {
    "ids": "car1_gps",
    "name": "Car 1 GPS",
    "mf": "Book2",
    "mdl": "ESP32 GPS Module",
    "sw": "1.0"
  },
  "o": {
    "name": "Home Assistant",
    "sw": "2026.3"
  },
  "cmps": {
    "latitude": {
      "p": "sensor",
      "unique_id": "car1_gps_latitude",
      "name": "Latitude",
      "state_topic": "car1/gps/raw",
      "value_template": "{{ value_json.latitude }}",
      "unit_of_measurement": ""
    },
    "longitude": {
      "p": "sensor",
      "unique_id": "car1_gps_longitude",
      "name": "Longitude",
      "state_topic": "car1/gps/raw",
      "value_template": "{{ value_json.longitude }}",
      "unit_of_measurement": ""
    },
    "speed": {
      "p": "sensor",
```

```
    "unique_id": "car1_gps_speed",
    "name": "Speed",
    "state_topic": "car1/gps/raw",
    "value_template": "{{ value_json.speed }}",
    "unit_of_measurement": "km/h"
  },
  "garage_approach": {
    "p": "binary_sensor",
    "unique_id": "car1_gps_garage_approach",
    "name": "Garage Approach",
    "state_topic": "car1/gps/garage_approach",
    "payload_on": "enter",
    "payload_off": "leave",
    "device_class": "presence"
  }
},
"availability_topic": "car1/gps/status",
"payload_available": "online",
"payload_not_available": "offline",
"qos": 0
}
```

This message will create the MQTT sensor `car1_gps` with the following entities:

- `sensor.car1_gps_latitude`
- `sensor.car1_gps_longitude`
- `sensor.car1_gps_speed`
- `binary_sensor.car1_gps_garage_approach`

Figure 8.11 shows MQTTX JSON payload issued to the topic `homeassistant/device/car1_gps/config`.

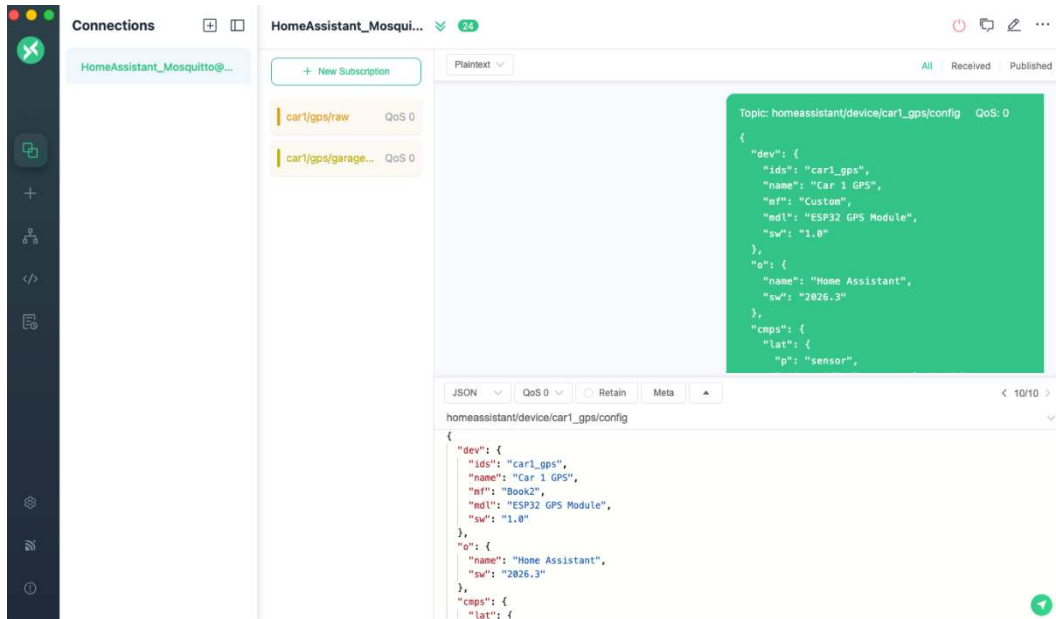


Figure 8.11 – MQTT ESP32 GPS based virtual sensor creation using MQTTX

Testing the GPS MQTT topics and automations using MQTTX

We will perform some tests to verify if the MQTT topics of the GPS sensor are working as expected.

Connectivity test

The first step is to confirm that communication between GPS Sensor and Home Assistant is working correctly. This can be done by publishing a simple payload message online to the topic `car1/gps/status`. When this message is received, the Car 1 GPS device should become available in Home Assistant. Publishing offline should mark the device as unavailable. This confirms that the MQTT broker, topic configuration, and discovery settings are functioning correctly.

GPS data validation

Next, a single GPS message can be published to verify that the sensor entities are correctly parsing the payload. The topic to publish the message at MQTTX is `car1/gps/raw` and a JSON message like the following produces the output in Home Assistant presented on *Figure 8.12*.

```
{
  "latitude": 34.8000,
  "longitude": -112.0093,
  "speed": 5
}
```

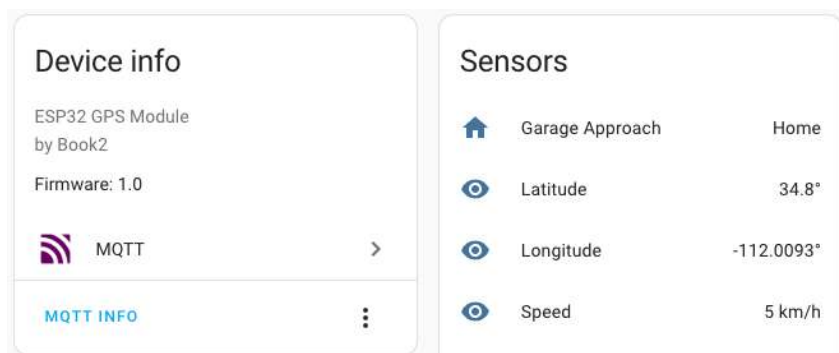


Figure 8.12 – MQTT ESP32 GPS based virtual sensor created in Home Assistant

Garage approach detection test

The next step is to validate the binary sensor responsible for detecting when the vehicle approaches the garage. The topic in this case will be `car1/gps/garage_approach` and the payload `enter` produces the output `Home` in Home Assistant. Sending the payload `leave` to the same topic we get in Home Assistant the status `Away` as presented on *Figure 8.13*.



Figure 8.13 – Garage approach detection test - Away

Automation testing

Once the sensor behavior is validated, the final step is to test the automation that opens the garage. Instead of controlling the garage directly through MQTT, the automation uses the

binary sensor as a trigger and executes an existing Home Assistant script presented on *Chapter 5 - Zigbee Garage Motor Automation*. I'm using the following automation:

```
alias: Test - Open garage from MQTTX (car1_gps)
triggers:
  - trigger: mqtt
    topic: car1/gps/garage_approach
conditions:
  - condition: template
    value_template: "{{ trigger.payload == 'enter' }}"
  - condition: state
    entity_id: binary_sensor.z_garage_gate_contact
    state: "off"
actions:
  - action: script.turn_on_off_garagemotor_script
    metadata: {}
    data: {}
mode: single
```

To test this automation, just publish enter to the topic car1/gps/garage_approach. If the system is correctly configured, the binary sensor will turn ON, the automation will trigger, and the garage door will begin opening. Publishing leave should not trigger any action. This completes our test with the virtual GPS sensor.

With the MQTT topics, payload structures, and automations now validated using MQTTX, the system is ready to transition from simulation to implementation. In the next section, we will build the actual GPS device using an ESP32 and a GPS module, and configure it to publish live location data to the MQTT broker. This step replaces the virtual sensor with a physical device, allowing the system to operate in a real environment while leveraging the same architecture and logic already tested.

Hands-on project: GPS tracking and location-based automations

In this section we will build a GPS-enabled IoT device using an ESP32 and a GPS module. The device continuously reads GPS sentences, parses them into geographic coordinates, and transmits them to Home Assistant through an MQTT broker. Once the data reaches Home Assistant, it can be used to trigger automations such as presence detection, notifications, or location-based actions. In our case as mentioned before, we will install in my car and it will be used to open and close my garage door when the car approaches or leaves the garage.

Note

This solution can also be implemented using ESPHome. The YAML code to implement is in GitHub. ESPHome implementation will limit the use of the solution within the Home Assistant range.

Hardware and system architecture

The Hardware wiring diagram for the solution to be implemented is presented on *Figure 8.14*.

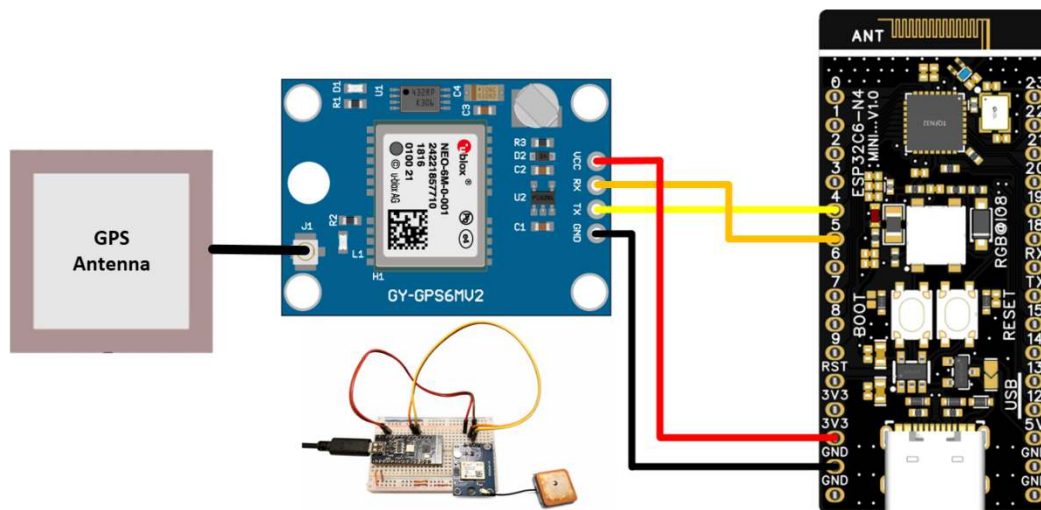


Figure 8.14 – MQTT GPS solution wiring diagram

We will use two main components to implement the solution: GPS Module GY-GPS6MV2 and an ESP32C6-Mini Wireless Communication module. The GPS module is based on u-Blox NEO-6M receiver and has the following features:

- GPS only constellation support
- 5Hz maximum update rate
- 3V3 or 5V Operating Voltage
- Communication port using 9600 baud rate UART
- Horizontal position accuracy of approximately 2.5 meters

The ESP32C6-Mini is a multi-communication board with the following features:

- 32-bit RISC-V single-core microprocessor, clocked at up to 160 MHz

- Supports Bluetooth 5.3 Low Energy (LE), Wi-Fi 6 (802.11ax) and Mesh networking.
- Fully compatible with Zigbee 3.0 and Thread 1.3
- Board supports a Multicolor WS-2812 LED

You can use different GNSS (Global Navigation Satellite System) modules like NEO-7M or NEO-M8N and other ESP32 modules to develop this solution.

The interconnection is very straightforward, needing only 2 wires between the GPS module and the ESP32 module besides the power supply (3V3 and GND). In the case of this example, pins IO4 and IO5 of the ESP32C6 Mini board are connected to the RX and TX pin of the GPS module.

In regard to the system architecture, *Figure 8.15* shows how the data flow is.

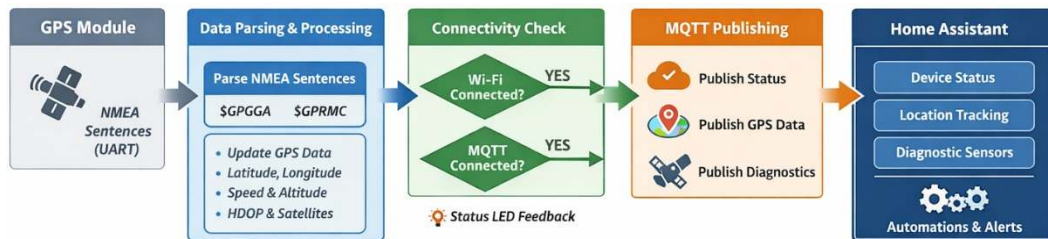


Figure 8.15 – System architecture

As presented in *Figure 8.15*, the GPS module sends GPS sentences to the ESP32C6 module using the UART interface. The GPS module has a small antenna connected to it to improve accuracy. It is good to emphasize that the GPS will work just outdoors or with some open sky visibility that can track satellites.

The ESP32C6 module implements an MQTT client and publishes GPS topics to the Mosquitto MQTT broker configured in Home Assistant whenever it is connected to the home Wi-Fi network. Outside the Wi-Fi coverage area, the device continues acquiring GPS data but cannot publish it until connectivity is restored. Home Assistant displays the received GPS data and triggers automations based on the reported location.

Next, before we describe the software implementation, I will describe a bit how a GPS works, how are the GPS sentences and what information we will get from it to help us to implement the solution.

GPS configuration, data and sentence

The NEO-6M GPS module works out of the box, which means that if you just connect the TX line to the EPS32 it will work providing data each second (1 Hz) using a 9600 baud rate. If you want to configure a different update rate like 5Hz or 10Hz, change the baud rate or reduce the GPS sentences sent, you will need the RX line connected to the ESP32. The ESP32 in this case

has to send commands to the NEO-6M to do it. For more information about the uBlox NEO-6M GPS module configuration checks its datasheet here: https://content.u-blox.com/sites/default/files/products/documents/NEO-6_DataSheet_%28GPS.G6-HW-09005%29.pdf

The GPS data parsed from the sentences will be the following:

- Latitude
- Longitude
- Altitude
- Speed
- HDOP (Horizontal Dilution of Precision)
- Number of Satellites

The HDOP is a measure of GPS accuracy. Lower HDOP values indicate better positional accuracy, while higher values indicate less reliable data. The Number of Satellites represents how many GPS satellites are currently being used to calculate the position. More satellites generally means more accuracy. The HDOP and Number of Satellites will support the diagnostics data for the **GPS fix**.

Note

GPS fix is an event when a GPS receiver successfully calculates its current position by receiving signals from multiple satellites.

The GPS modules use a standard communication protocol known as GPS sentences. These sentences are called NMEA (National Marine Electronics Association). Each sentence is a text message that begins with a \$ symbol and contains comma-separated data fields representing different types of navigation information.

These sentences provide the GPS data we need according to what we just mentioned. Common sentences we receive are:

- GGA: Provides fix data including latitude, longitude, altitude, number of satellites, and HDOP
- RMC: Provides recommended minimum data including position, speed, and time

These sentences must be parsed to extract the GPS data, allowing the system to obtain the location information and the diagnostics. This information will be useful for tracking and location-based automations implemented in Home Assistant.

We will now see how the software was implemented for this solution.

MQTT GPS solution sensor software implementation

This section describes the code implemented for the ESP32C6-Mini module. The code was developed using ESP-IDF V 6.0 compiler and VS-Code. We will break down our software implementation in the following subsections: MQTT topic design, System flow, Core software structure and Key function overview. The entire code including the configuration files is available on the book's GitHub repository. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface*.

MQTT topic design

The system is designed around three MQTT topics, each serving a specific role in the data flow between the ESP32 device and Home Assistant. All topics payloads are in JSON format.

Status Topic

Topic: car1/gps/status

This topic communicates the availability of the device. It is published as a retained message, ensuring that Home Assistant can immediately determine whether the device is online after a restart. The payload looks like the following:

```
{ "online": true }
```

This topic is essential for system reliability and is used by Home Assistant to manage entity availability.

Raw GPS Data

Topic: car1/gps/raw

This topic carries the primary location data used for automations and visualization. An example payload is:

```
{  
  "latitude": 36.302081,  
  "longitude": -116.8308675,  
  "speed": 1.16,  
  "altitude": 786.8  
}
```

Only valid GPS data is published, ensuring that invalid or incomplete readings are filtered at the source.

Diagnostics data

Topic: car1/gps/diagnostics

This topic provides quality and diagnostic information about the GPS fix. An example payload is the following:

```
{
  "satellites": 15,
  "hdop": 0.8
}
```

Separating diagnostics from raw data improves clarity and allows Home Assistant to categorize this information as diagnostic entities.

System flow

The system communication flow implemented in the ESP32 module covers the three middle blocks presented in *Figure 8.15*. The data parsing module reads NMEA sentences from the GPS module via UART. The incoming data is buffered and parsed line by line. The data processing module then parses the NMEA sentences to extract the desired information to include in the topics. Data is converted into structured values (latitude, longitude, speed, etc) and GPS data validity is evaluated.

The connectivity module verifies if the ESP32 can connect to Wi-Fi and to the MQTT broker with the credentials provided. If the module can connect to both Wi-Fi and Mosquitto, the MQTT publishing module is triggered publishing the JSON topics presented before. The system also implements a feedback circuit using the WS2812 multicolor LED that provides information about the connectivity, GPS Quality and signal accuracy.

Core software structure

The software is organized using FreeRTOS tasks, which is the multitasking Real Time Operating System (RTOS) that can be used with ESP-IDF. FreeRTOS allows the firmware to divide the application into independent execution loops, each dedicated to a specific responsibility. Instead of placing all logic inside a single while loop, the application separates GPS acquisition, MQTT publishing, and LED feedback into different tasks. This improves modularity, keeps the code easier to understand, and helps the system remain responsive even when several operations must run continuously at the same time.

In this project, the tasks are created in `app_main()` using `xTaskCreate()`. Three tasks are started: `gps_task`, `publish_task`, and `led_task`, each with its own stack size and priority. The GPS task is assigned the highest priority among them, followed by the publishing task, and then the LED task. *Table 8.4* outline some details about these three tasks.

Task	Responsibility	Update rate
gps_task	Read and parse GPS UART data	Continuous, 200ms UART timeout
publish_task	Publish MQTT messages	Every 5s
led_task	Update system status LED	Every 1s

Table 8.4 – FreeRTOS tasks implemented in the application

Note

In larger systems, or in designs with more complex shared data structures, synchronization mechanisms such as queues, semaphores, or mutexes could be introduced to improve robustness.

We will see an overview of the key functions used in the development of the system next.

Key function overview

Let's dig in the main functions used in the system implementation so you can have an idea about how it is implemented and modify it according to your needs.

GPS parsing functions

These are functions used to parsing the GPS sentences to the data we are looking for are the following:

```
static void parse_nmea(char *line);
static void parse_gga(char *sentence);
static void parse_rmc(char *sentence);
```

The parsed values are stored in a global structure called `gps_data_t`, making them accessible across tasks. The `gps_data_t` has the format as follows:

```
typedef struct {
    double latitude;
    double longitude;
    double speed_kmh;
    int satellites;
    double altitude;
    double hdop;
```

```
    bool valid_fix;  
} gps_data_t;
```

MQTT publishing functions

These functions format the data into JSON payloads and publish them to their respective topics. They are represented by the functions:

```
static void mqtt_publish_status(void);  
static void mqtt_publish_raw(void);  
static void mqtt_publish_diag(void);
```

The status topic is always published when the device is connected to the MQTT server. Raw and diagnostic data are published only when a valid GPS fix is available.

Connectivity management

These functions handle the Wi-Fi connection, MQTT client initialization, and connection state tracking. These are functions that implement these features:

```
static void wifi_init(void);  
static void mqtt_start(void);  
static void wifi_event_handler(...);  
static void mqtt_event_handler(...);
```

The system verifies connectivity before publishing data by checking the Wi-Fi connection through an event group bit and the MQTT connection using a boolean flag.

UART initialization

This function configures the UART interface used to receive NMEA sentences from the GPS module.

```
static void gps_uart_init(void);
```

The UART configuration is 9600 baud rate, 8 bits, no parity, 1 stop bit, and no flow control.

LED feedback system

The multicolor LED provides status diagnostics using the functions:

```
static void led_init(void);  
static void update_status_led(void);  
static void led_set_rgb(...);
```

The LED colors and status follow *Table 8.5*:

LED Color	Status	Number of Satellites	HDOP
Magenta	MQTT disconnected	-	-
Red	No GPS fix or poor accuracy	-	> 10
Yellow	Weak Signal	< 5	-
Blue	Good Signal	>= 5	<= 10
Green	Excellent Accuracy	> 10	< 2

Table 8.5 – LED colors x status

Some improvements in the code that I will let you figure out are: Adaptive publish rate which will provide faster updates when moving and slower when stationary, message filtering which will ignore GPS data when HDOP exceeds a threshold and payload optimization which can reduce message size for bandwidth efficiency.

With the ESP32 firmware now responsible for acquiring, processing, and publishing GPS data over MQTT, the focus shifts to how this information is consumed and used within Home Assistant. In the next section, we will finish the GPS solution implementation and explore how these MQTT messages are transformed into entities and leveraged to build location-based automations, enabling the system to react to the car movements.

Integrating GPS data into Home Assistant and building automations

In the last section when we tested the Virtual GPS using MQTTX, we used the `garage_approach` topic interact with Home Assistant and open the garage gate when a message enter was received. This requires the ESP32 device to calculate whether the vehicle was entering or leaving a predefined area. Although functional, this approach increases firmware complexity and logic to the device.

To simplify the embedded code and improve flexibility, this logic was removed from the ESP32 and delegated to Home Assistant. This design decision follows a key architectural principle to keep devices focused on data acquisition and move decision-making to the automation platform.

Also, as we added new parameters to the GPS sensor like altitude, number of satellites and HDOP, we had to change the MQTT discovery configuration. The new sensor configuration is

available on the book's GitHub repository. After sending the configuration to the topic `homeassistant/device/car1_gps/config` we will have the sensor configured in Home Assistant as shown in *Figure 8.16*. To access the repository link, follow the steps in the "Download the example code files" section in the *Preface*.

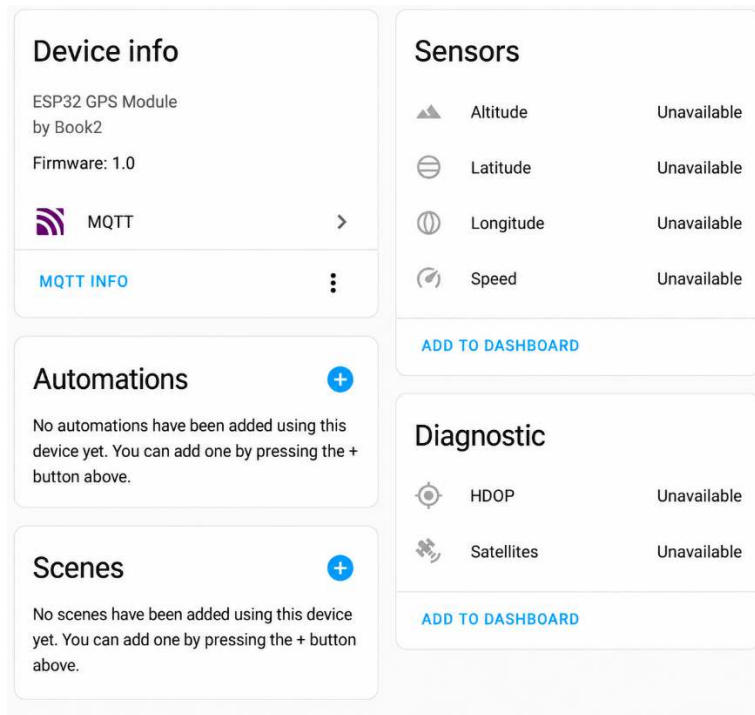


Figure 8.16 – New GPS sensor configured in Home Assistant - Disabled

You can see in *Figure 8.16* the data is unavailable. This is how the sensor will look if you do not have the hardware device connected and working.

Before we create the automation, we will have to create a helper that will be used as a latch to avoid the automation being triggered multiple times when the car is in the garage. We will call this latch `car1_arrived` and you will have to use the helper type toggle to create it. To create a helper, you will use Settings | Devices & services | Helpers.

The Automation we will do now will rely on the coordinates received from the GPS and verify if they are less than 20 meters of the home zone coordinates. If they are and the garage gate is closed and the helper `car1_arrived` is off then the automation executes the action to run the

script that open the garage gate. The automation will also turn on the helper latch `car1_arrived`. The YAML code below lists the automation created.

```
alias: Open Garage - GPS Based (car1_gps)

trigger:
  - platform: mqtt
    topic: car1/gps/raw

conditions:
  - condition: template
    value_template: >
      {% set lat = trigger.payload_json.latitude %}
      {% set lon = trigger.payload_json.longitude %}
      {{ distance(lat, lon, state_attr('zone.home', 'latitude'),
state_attr('zone.home', 'longitude')) < 0.02 }}

  - condition: state
    entity_id: binary_sensor.z_garage_gate_contact
    state: "off"

  - condition: state
    entity_id: input_boolean.car1_arrived
    state: "off"

actions:
  - action: script.turn_on_off_garagemotor_script
    metadata: {}
    data: {}
  - action: input_boolean.turn_on
    metadata: {}
    data: {}
    target:
      entity_id: input_boolean.car1_arrived

mode: single
```

We will have to create another automation to reset the helper `car1_arrived` when the car is leaving the garage. The YAML code to create this automation is the following:

```
alias: Reset Garage Arrival Latch (car1_gps)
description: Resets the arrival latch when the car leaves the Home Zone
```

```
triggers:
  - trigger: mqtt
    topic: car1/gps/raw
conditions:
  - condition: template
    value_template: >
      {% set lat = trigger.payload_json.latitude %}
      {% set lon = trigger.payload_json.longitude %}
      {{ distance(lat, lon, state_attr('zone.home', 'latitude'),
state_attr('zone.home', 'longitude')) >= 0.03 }}
  - condition: state
    entity_id: input_boolean.car1_arrived
    state: "on"
actions:
  - action: input_boolean.turn_off
    metadata: {}
    data: {}
    target:
      entity_id: input_boolean.car1_arrived
mode: single
```

Note that we will use 30 meters away from the home zone to set the car1_arrived helper off. These distances were tested using my home zone coordinates which had a radius of 11 meters.

Note

You will have to calibrate the close (20m), away (30m) distances and the home zone radius (11m) depending on the size of your home and your Wi-Fi coverage so the automations can work properly.

Figure 8.17 shows the points I used for testing the automation close to my home.

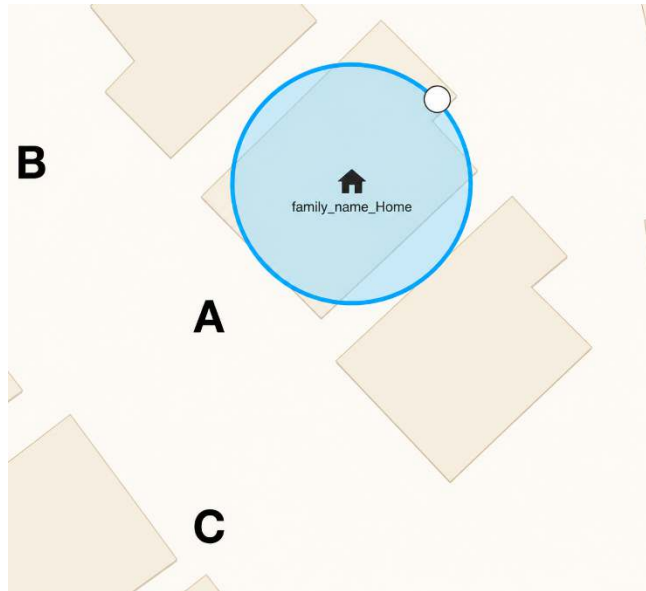


Figure 8.17 – Points tested for automations

To test these automations, follow these steps:

1. Assuming you are in your garage, to test if the automation Open Garage - GPS Based (car1_gps) works install the ESP32 GPS device in your car by connecting it to the car USB port to power up and place it making sure you get enough sky visibility.
2. While the device is connecting to the MQTT broker, the LED will be magenta and if it gets a number of satellites greater than ten and HDOP less than two, after it connects to the MQTT broker the LED will change to green.
3. Now, drive away from your home so you will observe that the LED turns to magenta again, meaning your home Wi-Fi coverage is gone. These location points in my home were approximately the B and C in Figure 8.14. At this time, the automation Reset Garage Arrival Latch (car1_gps) should be activated by resetting the helper car1_arrived.
4. Return home slowly, getting your car closer to the house.
5. When the GPS reaches the Wi-Fi area in your home and connects to the server, it will already get a fix so it will transmit the coordinates and if within the threshold distance of the home zone area, the automation Open Garage - GPS Based (car1_gps) will be triggered, and the garage gate will be opened. This was the location point A in Figure 8.14.

If the distance adjustment becomes difficult, you can think of adding the combination of distance and speed. The speed should be reduced when entering home and increased when leaving home.

Note

This implementation is intended as a proof-of-concept to demonstrate MQTT communication and location-based automations. Because it relies on the home Wi-Fi network, the ESP32 publishes GPS data only after reconnecting when entering the Wi-Fi coverage area. In a production system, improvements such as cellular connectivity, assisted GPS, buffering GPS data while offline, or combining multiple sensors and detection methods could provide faster and more reliable operation.

This implementation shows how a GPS-enabled ESP32 device can reliably publish location data using MQTT and integrate it into Home Assistant for automation. By keeping the device focused on data acquisition and moving decision-making to the platform, the solution remains simple, flexible, and scalable for future extensions.

Summary

This chapter introduced the MQTT ecosystem as a communication layer for advanced smart home automation systems. It began by presenting the core MQTT concepts, including brokers, clients, topics, and Quality of Service (QoS) levels. Building on these concepts, the chapter guided you through the setup and secure configuration of an MQTT broker using Mosquitto.

The chapter then focused on MQTT tools. You learned how to use MQTT Explorer and MQTTX by using practical examples. A key part of the chapter was the hands-on project, where MQTT was used to transmit GPS data from a device to Home Assistant. This project demonstrated how the same MQTT architecture can be used to integrate a wide variety of custom IoT devices with Home Assistant, providing a flexible and scalable foundation for future smart home projects.

In *Chapter 9, Building Touchscreen Display and Camera Devices*, these concepts will be extended by developing a complete embedded system using ESPHome and MQTT. You will learn how to integrate a touchscreen display and camera with an ESP32, creating an interactive smart home device capable of both visual feedback and communication with Home Assistant.

9

Building Touchscreen Display and Camera Devices

In the previous two chapters we explored ESPHome and the MQTT Ecosystem to implement custom device solutions. Now, we take the next step by combining everything learned so far in these two chapters into fully interactive smart devices.

This chapter focuses on building visual and interactive devices using ESP32, touchscreen displays and cameras. Unlike traditional sensors that only send data, this device provides real-time feedback and user interaction directly at the edge, reducing dependency on dashboards and improving usability.

You will begin by understanding the various types of displays and touchscreens compatible with ESP32. The sequence will then guide you through the hardware interfacing of these components and, crucially, the ESPHome configuration required to control them. You will learn how to stream camera data to Home Assistant for live viewing. The chapter will wrap up by empowering you to build standalone smart ESP32 cameras with visual feedback.

By the end of this chapter, you will learn to interface ESP32 with a TFT display technology and touchscreen, configure ESPHome to control displays and create custom UI elements, integrate raw camera modules with ESP32, stream camera data to Home Assistant for live viewing, and develop interactive smart home devices with visual feedback and camera capabilities.

We will cover the following topics in this chapter:

- Main project overview and system architecture
- Hardware interfacing: connecting displays, touchscreens, and cameras to ESP32
- Building a TFT touchscreen software application using ESPHome and MQTT

- Integrating cameras with ESPHome and Home Assistant
- ESP32-S3 camera as an independent device

Technical requirements

To reproduce the examples in this chapter, you should have the following:

Software Requirements:

- A working Home Assistant installation with an MQTT broker configured.
- Basic knowledge of ESPHome, YAML, and MQTT for device integration and automation.
- Familiarity with ESP-IDF and Visual Studio Code if you plan to build, modify, or compile the standalone ESP32-S3 camera firmware.

Hardware Requirements:

The chapter presents two independent hands-on examples, each using a different ESP32 platform:

- ESP32 with a 3.5-inch TFT touchscreen display (used in the Smart Casa Monitor project to demonstrate ESPHome, LVGL, and MQTT integration).
- ESP32-S3-CAM development board with an OV3660 camera sensor (used as a separate example to demonstrate a standalone camera solution integrated with Home Assistant).

You do not need both boards to follow the chapter. Each project can be completed independently using the corresponding hardware.

You can download the code for all examples in this book by following the instructions in the **Download the example code files** section in the *Preface* of this book.

This chapter's code files are included in the downloadable code bundle.

Main project overview and system architecture

The main project described in this chapter came from a practical need to have a simple and fast way to control key switches in my home without relying on voice assistants or my cell phone. The goal was to interact with some of the most used devices, just by touching a screen, without delays or extra steps.

At the same time, I've been interested in expanding the use of custom cameras around my home. That led to the idea of combining both concepts into one device bringing together a

touchscreen interface for control and a camera feed for monitoring, all connected to Home Assistant.

System architecture overview

The system, called Smart Casa Monitor, is composed of four main subsystems:

- ESP32 processing unit: Acts as the central controller, handling communication, logic, and integration with Home Assistant
- Display Interface: Composed of TFT Display, it is responsible for rendering information and UI elements
- Touch Input: Enables direct user interaction with the system
- Camera Streaming: Provides visual monitoring

This device acts as both an input and output system. It captures user interaction through the touchscreen and acquires visual data from the camera. On the output side, it displays information on the screen and streams the camera feed in real time. *Figure 9.1* shows the system architecture.

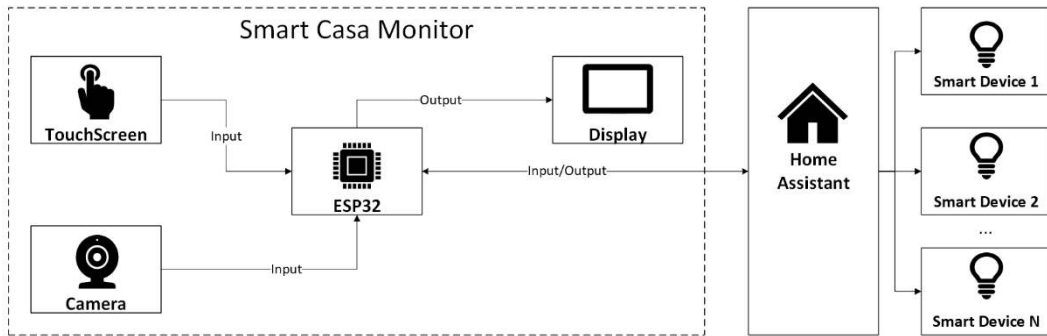


Figure 9.1 – System architecture

The system shown in *Figure 9.1* works through a continuous loop between the user, the Smart Casa Monitor device, and Home Assistant. The user interacts with the touchscreen, and the ESP32 processes this input locally before sending the data to Home Assistant using MQTT. The MQTT Server installed on Home Assistant server then sends the commands to the MQTT devices and updates their states. At the same time, the integrated camera provides a live stream that can be used for monitoring.

Now that we have an idea about the system architecture, let's understand some technologies for displays and touchscreens and then in the next section we can dive more in the Hardware components used in this project.

Understanding display and touch technologies

Currently there are several display options that can be used in Home Automation devices. *Table 9.1* summarizes the display technologies available and commonly used.

Type	Pros	Cons	Typical Use Case
OLE D	Very low power (typically 20–80 mW for mostly dark screens), excellent contrast (true blacks)	Burn-in risk with static images, higher cost	Status indicators, small displays, battery-powered devices
TFT LCD	Fast refresh (30–60 fps), good color reproduction, affordable	Moderate power consumption (typically 300–800 mW with backlight)	Dashboards, touch interfaces, real-time user interfaces
IPS TFT LCD	Wide viewing angles (typically 170°–178°), accurate colors, good brightness	Higher power consumption (typically 500 mW–1.5 W depending on backlight size)	Premium touch panels, wall-mounted control panels, rich graphical interfaces
E- Paper	Extremely low power (power used mainly during screen updates), readable in sunlight	Slow refresh (typically 0.5–2 s), limited animation capability	Status displays, calendars, labels, price tags

Table 9.1 – Summary of display technologies

For this project, a TFT SPI display is selected because it supports real-time updates, which are essential for responsive user interaction, and it enables rich graphical interfaces needed for dynamic dashboards. It is fully compatible with the ESP32 and ESPHome ecosystem, making integration straightforward.

Regarding the touchscreen, there are basically two technologies available: resistive and capacitive. Resistive touchscreens detect pressure, so they work with fingers and gloves, but they are less responsive and do not support multi-touch. Capacitive touchscreens detect electrical changes, offering higher sensitivity and multi-touch support, but they usually require direct finger contact and are more expensive.

We will use a capacitive touch because it is more accurate and responsive compared to the resistive touch. Pressure is not required to activate it, so it has better user experience for home usage.

Hardware interfacing: connecting displays, touchscreens, and cameras to ESP32

The best hardware platform is the one that integrates all four subsystems commented in the last section. The best option found was the Waveshare ESP32-S3-Touch-LCD-3.5 which has the following features:

- ESP32-S3R8 Dual-core Xtensa LX7 processor up to 240 MHz including Built-in Wi-Fi (2.4 GHz) and Bluetooth 5 (LE), 512 KB SRAM + 384 KB ROM, 8 MB PSRAM + 16 MB Flash.
- 3.5" TFT LCD with a resolution of 320 x 480 pixels, color depth of 262K colors, driver based on the ST7796 controller and SPI interface.
- Capacitive touch screen based on the FT6336 controller chip and interface I2C.
- Camera integration capability supporting sensors OV2640, OV3660 and OV5640.

The board offers many more features beyond those listed above, as you can check in this link: <https://www.waveshare.com/product/esp32-s3-touch-lcd-3.5.htm>. *Figure 9.2* shows a picture from the back side of the ESP32-S3-Touch-LCD-3.5 board.

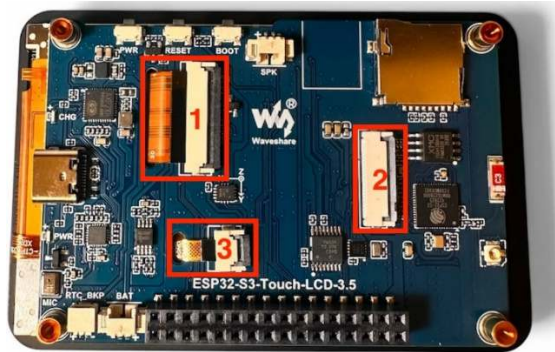


Figure 9.2 – ESP32-S3-Touch-LCD-3.5 board used in the project

The three red rectangles marked on *Figure 9.2* represent the interface connector between the ESP32-S3 and display (1), camera (2), and the touchscreen (3). This means that the only external devices required to be connected to the board are the camera sensor in the connector 2 and the power supply via USB-C connector. This simplifies a lot about wiring connectivity and lets us focus on software development for the solution.

As mentioned, the next section will discuss the software development, including how ESPHome handles the display and touchscreen, and how we can implement the user interface using it.

Building a TFT touchscreen software application using ESPHome and MQTT

In this section, we will develop a touchscreen application for the Waveshare ESP32-S3 3.5-inch display using ESPHome. The display will act as a smart home control panel that communicates with Home Assistant through MQTT. Before implementing the application, it is useful to define its main features and user interface.

The application has the following functional requirements:

- Eight configurable touchscreen buttons, each associated with an MQTT topic.
- Publish ON/OFF MQTT commands when a button is pressed.
- Two main control pages supporting up to eight configurable buttons.
- A navigation bar with Previous, Settings, and Next buttons.
- A Settings page to configure button names and MQTT topics.
- Local storage of configuration parameters.
- Wi-Fi and MQTT connection status indicators.
- Clock synchronized with Home Assistant.
- Automatic screen dimming after inactivity.
- Automatic return from the Settings page after a timeout.

Figure 9.3 shows the proposed screen layout that will be implemented throughout this section.



Figure 9.3 – Application screen prototype mockup

Before diving into the implementation details, it is helpful to understand the overall development sequence of the application. *Figure 9.4* presents the workflow used throughout this project, starting with the hardware configuration, followed by the user interface design and implementation, application logic development, user experience enhancements, and finally deployment and testing on the ESP32. The following sections explore each of these steps in detail.

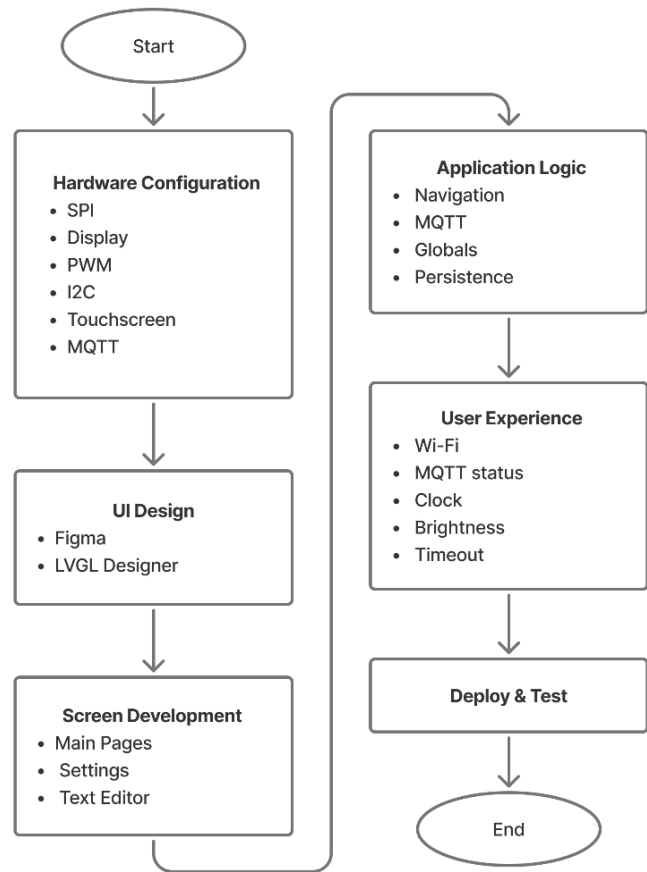


Figure 9.4 – Smart Casa Monitor development process flowchart

Configuring the display and touchscreen in ESPHome

By consulting the Waveshare ESP32-S3 3.5-inch board schematics at <https://files.waveshare.com/wiki/ESP32-S3-Touch-LCD-3.5/ESP32-S3-Touch-LCD-3.5-Schematic.pdf> we can have the interconnection diagram between the ESP32 and the display and touchscreen presented in *Table 9.2*.

Interface	Signal	ESP32-S3 Pin	Device / controller
Display	SCL / SPI CLK	GPIO5	ST7796 display - SPI
Display	SDA / SPI MOSI	GPIO1	ST7796 display - SPI

Interface	Signal	ESP32-S3 Pin	Device / controller
Display	SDO / SPI MISO	GPIO2	ST7796 display - SPI
Display	DC	GPIO3	ST7796 display
Display	Backlight PWM	GPIO6	LCD backlight circuit
Touchscreen	I2C SCL	GPIO7	FT6336 touch controller - I2C
Touchscreen	I2C SDA	GPIO8	FT6336 touch controller - I2C

Table 9.2 – Interconnection signals – ESP32 x display and touchscreen

Using Table 9.2, we can create the following ESPHome YAML configuration for the Display.

```

spi:
  - id: lcd_spi
    clk_pin: GPIO5
    mosi_pin: GPIO1
    miso_pin: GPIO2

display:
  - platform: mipi_spi
    id: main_display
    model: ST7796
    spi_id: lcd_spi
    dc_pin: GPIO3
    dimensions:
      width: 320
      height: 480

```

We basically have to define a `spi:` configuration using the pins provided on Table 9.2 and other required `display:` configurations like `platform`, `model` and `dimensions`. For more ESPHome configuration parameters for the display MIPI SPI driver check here: https://esphome.io/components/display/mipi_spi/

The LCD backlight ESPHome configuration is done configuring an `output: pin` to generate a Pulse Width Modulation (PWM) and then using this output pin in a `light: configuration`. The YAML code to configure the backlight is the following:

```
output:
  - platform: ledc
    pin: GPIO6
    id: backlight_pwm

light:
  - platform: monochromatic
    id: display_backlight
    output: backlight_pwm
    restore_mode: ALWAYS_ON
```

The touchscreen configuration is done by configuring an `i2c: bus` first and then using the `touchscreen: directive` to fill the required parameters `platform` as `ft63x6` and `display:`. The touchscreen `i2c` configuration is linked to `i2c` bus using the parameter `i2c_id`. The YAML code to configure the `i2c` and `touchscreen` is presented below:

```
i2c:
  - id: bus_a
    sda: GPIO8
    scl: GPIO7

touchscreen:
  - platform: ft63x6
    id: main_touch
    display: main_display
    i2c_id: bus_a
```

After the display and touchscreen we will configure the MQTT parameters in the next subsection.

Configuring the MQTT for the application

The MQTT configuration is straightforward and defines the connection parameters for the broker. It is done by providing information like server, port, user and password. The YAML we will use for this application is the following:

```
mqtt:
  id: mqtt_client
  broker: "192.168.8.190"
  port: 1883
  username: "your_name"
  password: "your_password"
  discovery: false
  reboot_timeout: 0s
  log_topic: null
```

We will configure directive discovery as false to disable automatic entity discovery. Also we will disable the application to reboot if no MQTT connection exists using `reboot_timeout: 0s`. Using the directive `log_topic: null` we disable the application to send logs to MQTT.

Building the user interface with LVGL support

To build the user interface, we will use LVGL support in ESPHome. LVGL (<https://lvgl.io/>), which stands for Light and Versatile Graphics Library, is an open-source framework used to create rich graphical user interfaces on microcontrollers and embedded displays. Instead of drawing every pixel, shape, or interaction manually, LVGL provides building blocks such as buttons, labels, sliders, charts, and containers.

A structured and iterative approach was followed to develop the LVGL interface. It started from the visual mockup design created in *Figure 9.3* using Figma (<http://www.figma.com>). Next step was to translate the UI layout into a working prototype. The ESPHome LVGL Designer tool (<https://lvgl.espbboards.dev>) was used for that. This tool allowed the creation of the screens visually, matching the layout, spacing, and components defined in the mockup. After the creation you can export the YAML code and Copy/Paste to ESPHome. *Figure 9.5* shows the Functional UI creation using the ESPHome LVGL Designer tool.

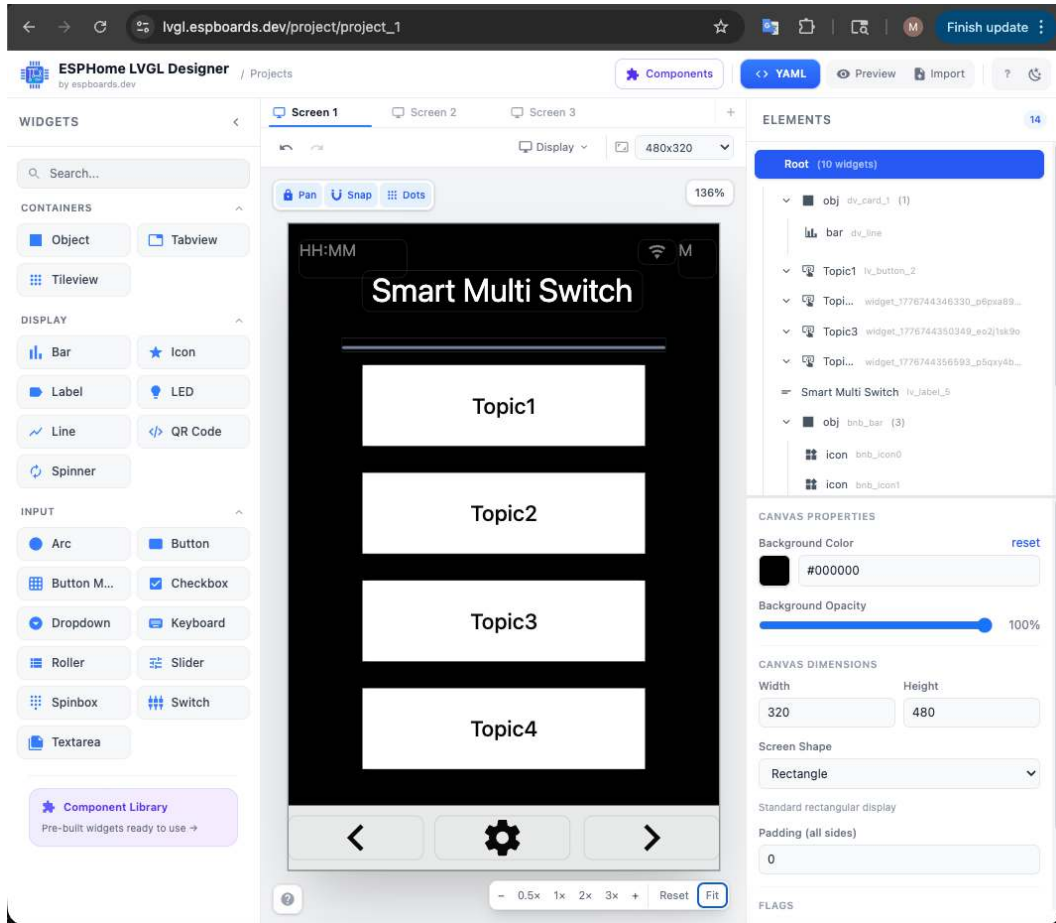


Figure 9.5 – ESPHome LVGL Designer tool

The development sequence adopted to develop the UI portion of the ESPHome YAML code was the following:

1. Create UI mockup in Figma shown in *Figure 9.3*.
2. Build screen layouts using ESPHome LVGL Designer as shown in *Figure 9.5*.
3. Implement navigation between pages using buttons.
4. Add logic to edit button names and configure MQTT topics.
5. Implement MQTT publish functionality for each button.
6. Store configuration using global variables for persistence.

7. Add status indicators icons for Wi-Fi and MQTT and time.
8. Implement display brightness control after 2 minutes of inactivity and automatic return to the main screen after 5 minutes in settings.

The next subsections will discuss some of these steps.

Building screen layouts using ESPHome LVGL Designer

In total, 5 LVGL screens were developed for the project. These screens are named as follows:

- Main Screen – Page 1 (`page_1`): Displays the first set of four control buttons along with the navigation bar.
- Main Screen – Page 2 (`page_2`): Displays the second set of four control buttons and allows navigation back to Page 1.
- Settings Screen – Topics 1 to 4 (`page_topic_settings_1`): Allows configuration of button names and MQTT topics for the first four buttons.
- Settings Screen – Topics 5 to 8 (`page_topic_settings_2`): Allows configuration of button names and MQTT topics for the remaining four buttons.
- Text Editor Screen (`page_text_editor`): Dedicated screen used to edit individual names or MQTT topics fields using the on-screen keyboard.

Development of the screens is very intuitive using the ESPHome LVGL Designer tool. You have to set up the display size, which in this case will be 320x480. After that, you have to drag and drop the widgets from the toolbar, located on the left side of the screen shown in *Figure 9.5*. By clicking on each widget, you can configure its parameters in the toolbar located on the right of the screen.

Of the 5 screens created, the Main Screens (Page 1 and Page 2) and the Settings Screens (Topics 1 to 4 and Topics 5 to 8) are very similar, just handling different content to accommodate their respective topics. The Text Editor screen serves both pairs of pages. *Figure 9.6* shows 3 of the 5 screens created using the ESPHome LVGL Designer tool.

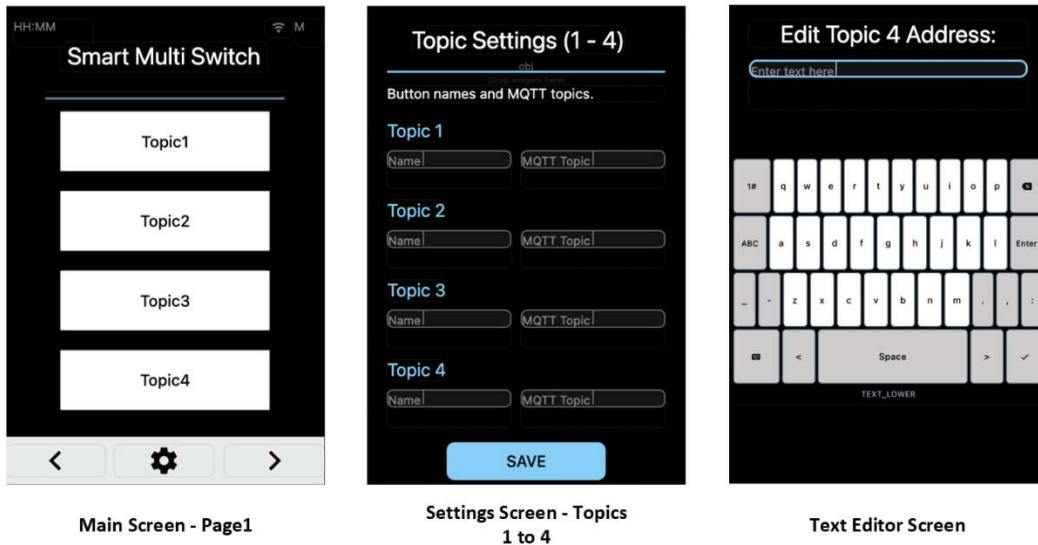


Figure 9.6 – Three screens created using ESPHome LVGL Designer tool

After creating each screen, click the Export drop-down button in the upper-right corner of the ESPHome LVGL Designer and select Download .yaml to download the generated configuration. Copy the exported YAML into the LVGL section of your ESPHome project and repeat the process for each screen you create.

The next subsection will show how navigation works among the screens and how to implement it in ESPHome.

Implementing navigation between pages

After importing the YAML generated by the ESPHome LVGL Designer into ESPHome, the next step is to establish the navigation between the application screens. *Figure 9.7* illustrates how the user moves between the two main pages, the settings pages, and the shared text editor.

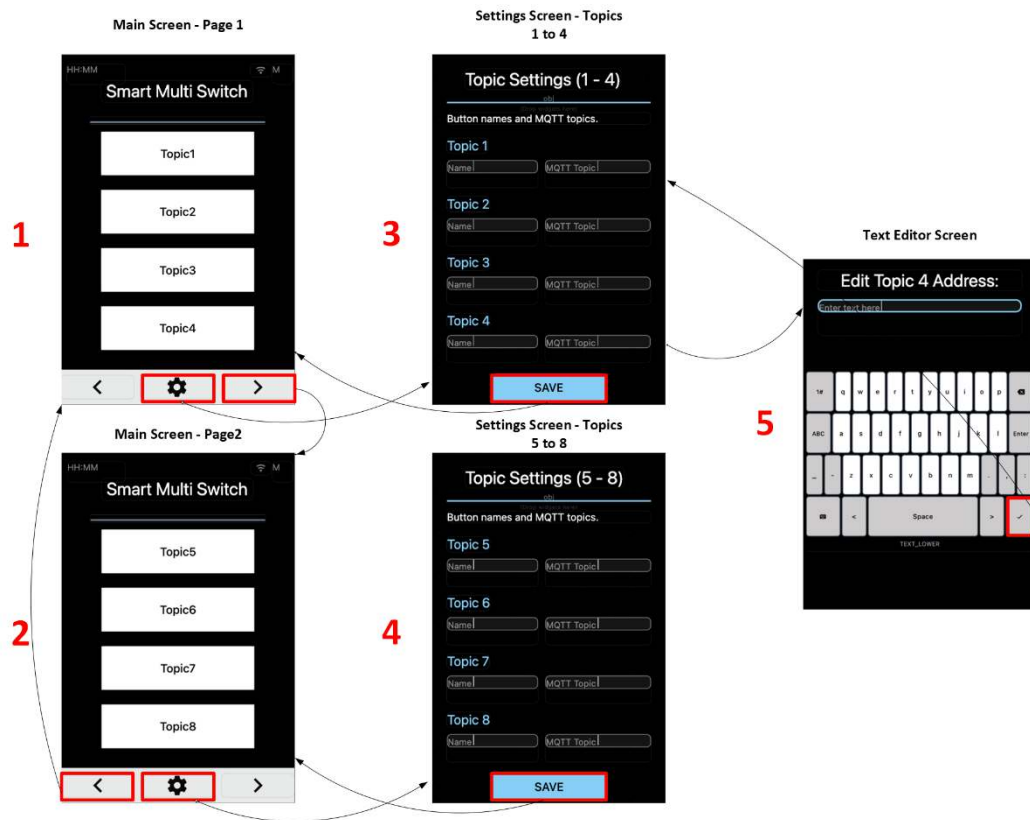


Figure 9.7 – Screens navigation

Navigating between the main and settings pages

The **Main Screen – Page 1** (Figure 9.7 – number 1) is displayed when the application starts. Pressing the > button opens **Main Screen – Page 2** (Figure 9.7 – number 2), while pressing the < button on Page 2 returns to Page 1. This navigation is implemented using the `lvgl.page.show` action, as shown below.

```
- button:
...
    on_press:
then:
    - lvgl.page.show:
        id: page_2
...
```

```

- button:

...
  on_press:
  then:
    - lvgl.page.show:
      id: page_1

```

The same approach is used to navigate to the settings pages. Pressing the gear icon on Page 1 opens **Settings Screen – Topics 1 to 4** (Figure 9.7 – number 3), while pressing the gear icon on Page 2 opens Settings Screen – Topics 5 to 8 (Figure 9.7 – number 4). Part of the ESPHome YAML code that implements this feature is presented as follows.

```

- button:
  id: nav_settings_page_1
  ...
  on_press:
    then:
      - lvgl.page.show:
        id: page_topic_settings_1
...
- button:
  id: nav_settings_page_2
  ...
  on_press:
    then:
      - lvgl.page.show:
        id: page_topic_settings_2

```

Each settings page contains eight editable fields, allowing the user to configure four button names and their corresponding MQTT topics.

Opening the shared text editor

Instead of creating a separate editing screen for every field, the application uses a single Text Editor screen (Figure 9.6 – number 5). When the user selects any editable field, the current value is copied into the editor, the on-screen keyboard is displayed, and two global variables are updated:

- `active_edit_field` identifies which field is being edited.
- `return_settings_page` stores the settings page that opened the editor so the application can return to it afterward.

The following example shows how the first topic name field opens the shared editor.

```
- id: page_topic_settings_1
  - textarea:
      id: txt_topic_1_name
      ...
      on_focus:
        then:
          - lambda: |-
              id(active_edit_field) = 1;
              id(return_settings_page) = 1;
              lv_label_set_text(id(lbl_text_editor_title), "Edit Topic 1
Name");

              lv_textarea_set_text(id(txt_editor_value),
lv_textarea_get_text(id(txt_topic_1_name)));
          - lvgl.keyboard.update:
              id: editor_keyboard
              textarea: txt_editor_value
          - lvgl.page.show:
              id: page_text_editor
```

Returning the Edited Value

When the user presses the check button on the on-screen keyboard, the edited text is copied back to the appropriate field. A switch statement uses the value stored in `active_edit_field` to determine which text box should receive the updated value. After the update, the application returns to the settings page that originally opened the editor.

The simplified example below shows the process.

```
- id: page_text_editor
  - keyboard:
      id: editor_keyboard
      ...
      on_ready:
        then:
          - lambda: |-
              const char *edited =
                lv_textarea_get_text(id(txt_editor_value));
              switch (id(active_edit_field)) {
                case 1:
                  lv_textarea_set_text(
```

```

        id(txt_topic_1_name), edited);
        break;

        /* Remaining fields follow the same pattern */
    }

- if:
    condition:
        lambda: 'return id(return_settings_page) == 1;'
    then:
        - lvgl.page.show:
            id: page_topic_settings_1
    else:
        - lvgl.page.show:
            id: page_topic_settings_2

```

The two global variables used by the navigation logic are declared as follows:

```

globals:
- id: active_edit_field
  type: int
  restore_value: no
  initial_value: "1"

- id: return_settings_page
  type: int
  restore_value: no
  initial_value: "1"

```

Using a single text editor significantly reduces the amount of duplicated code while providing a consistent editing experience for all button names and MQTT topic addresses. In the next subsection, we will extend the application by implementing the MQTT publish functionality associated with each touchscreen button.

Implementing MQTT publish functionality on buttons

At this point in the application, we can navigate through all screens and edit the MQTT topic names and addresses. We will now learn how to add code so that, when a button is pressed, it

changes color and sends an MQTT command to the MQTT server, which distributes it to the switch associated with that topic. We will do that using the following code:

```
on_press:
  then:
    - lambda: |-
        id(topic_1_state) = !id(topic_1_state);

        const char *topic = id(topic_1_address).c_str();
        const char *payload = id(topic_1_state) ? "ON" : "OFF";

        lv_obj_set_style_bg_color(
            id(btn_topic_1),
            lv_color_hex(id(topic_1_state) ? 0x87CEFA : 0xFFFFFFFF),
            LV_PART_MAIN
        );

        id(mqtt_client).publish(topic, payload);
```

We have two global variables defined: `topic_1_state` and `topic_1_address`. The `topic_1_state` is a boolean type variable that holds the MQTT topic address state and `topic_1_address` is a string type variable that stores the MQTT topic address.

The code presented toggles the `topic_1_state` variable each time the button is pressed. It also converts the address and state global variables to C-style string (`const *char`) and formats the payload as ON or OFF depending on the state defined by the `topic_1_state` variable.

The LVGL function `lv_obj_set_style_bg_color` sets the button color to white (0xFFFFFFFF) or a variation of blue (0x87CEFA). The code snippet finalizes with a function to publish the payload to the MQTT topic using the function `id(mqtt_client).publish(topic, payload)`.

The code snippet is the same for all eight buttons just changing the `topic_x_state` and `topic_x_address` variable names where `x` varies from 1 to 8.

The next subsection will detail a little more about the use of some global variables presented in this subsection and how they support storing and retrieving configuration values.

Storing configuration using global variables for persistence

Some global variables were created to store the values of the information associated with the buttons. These variables are the `topic_x_state`, `topic_x_address` mentioned in the last

subsection and `topic_x_name`. The `topic_x_name` is used to store the name displayed in the buttons.

Each time the SAVE button is clicked on each settings page, code is executed to collect the information entered in the text fields for the topic name and topic address, and store them in the variables `topic_x_name` and `topic_x_address`. The ESPHome YAML code that implements this is the following:

```
id(topic_x_name) = lv_textarea_get_text(id(txt_topic_x_name));
id(topic_x_address) = lv_textarea_get_text(id(txt_topic_x_address));
```

What also happens when the SAVE button is clicked is that the displayed button name is updated using the following YAML code:

```
lv_label_set_text(id(lbl_topic_x), id(topic_x_name).c_str());
```

When the unit is booting the button names and text fields are populated with the previous values stored in the global variables mentioned using the code:

```
lv_label_set_text(id(lbl_topic_x), id(topic_x_name).c_str());
lv_textarea_set_text(id(txt_topic_x_name), id(topic_x_name).c_str());
lv_textarea_set_text(id(txt_topic_x_address), id(topic_x_address).c_str());
```

The next section will present how the Wi-Fi and MQTT icons are presented on the screen.

Creating status indicator icons for Wi-Fi, MQTT, and clock

With the features created so far, the application has the minimal functionality to operate as it is supposed to be. The extra features described in this and in the next subsection are considered extras to improve usability and user experience.

The Wi-Fi and MQTT icons are created in the top right corner of the screen but when initially created they are hidden and are just shown if the unit succeeds in connecting to the Wi-Fi network and the MQTT server. In total, four icons are created — one for Wi-Fi and one for MQTT on each main page. As an example, the code used to show and hide the Wi-Fi icon is the following:

```
on_connect:
  then:
    - lvgl.widget.show: icon_mqtt_page_1
    - lvgl.widget.show: icon_mqtt_page_2
on_disconnect:
```

```

then:
  - lvgl.widget.hide: icon_mqtt_page_1
  - lvgl.widget.hide: icon_mqtt_page_2

```

These icons help with diagnostics — for example, when a button is pressed but the switch doesn't turn on or off. If the Wi-Fi or MQTT is not connected, the commands will not be sent.

The clock feature displays the current time in the top left corner of the screen, and it is updated each minute. It helps the user know what time it is and also supports diagnostics, to see if the system is locked when the time is not updated. The code responsible to implement this functionality is the following:

```

time:
  - platform: homeassistant
    id: ha_time

interval:
  - interval: 60s
    then:
      - lambda: |-
          auto now = id(ha_time).now();
          if (now.is_valid()) {
              char time_str[6];
              snprintf(time_str, sizeof(time_str), "%02d:%02d", now.hour,
now.minute);

              lv_label_set_text(id(lbl_time_page_1), time_str);
              lv_label_set_text(id(lbl_time_page_2), time_str);
          }

```

The next subsection will show how to use a timer feature to control display brightness and to return to the main screen due to inactivity.

Implement a timer feature to display brightness and screen inactivity control

Two other remaining extra features were implemented for the application. The first feature reduces the display brightness to 20% after 2 minutes. This feature has several advantages, such as power consumption reduction, heat reduction, long term reliability improvement, and user comfort.

The brightness control feature is implemented using the following ESPHome script directive.

```
script:
  - id: reset_screen_timeout
    mode: restart
    then:
      - light.turn_on:
          id: display_backlight
          brightness: 100%
      - delay: 2min
      - light.turn_on:
          id: display_backlight
          brightness: 20%
```

This script runs at two moments in the implemented code: at boot, when the unit starts, and when the user touches the screen, using the `on_touch` event.

The screen inactivity control is implemented similarly to the brightness control using a script called `settings_timeout`, which triggers a five-minute timer each time a textarea or a button, with the exception of the Save button, is clicked. After 5 minutes, if no touchscreen activity is detected, the screen backlight brightness will be reduced to 20% and the application will return to Main Screen Page 1.

Other extra features can be implemented to improve the application reliability like a subscription to the MQTT topic related to the switch where the payload was sent to verify the status of the command sent. This will be a future implementation.

Figure 9.8 shows the real application deployed in the display using some of the screens developed according to *Figure 9.6*. *Video 9.1* (<https://packt.link/UbDFQ>) demonstrates the Smart Casa Monitor user interface in operation, showing navigation between screens, configuration of MQTT topics, and the use of the on-screen keyboard to edit application settings.

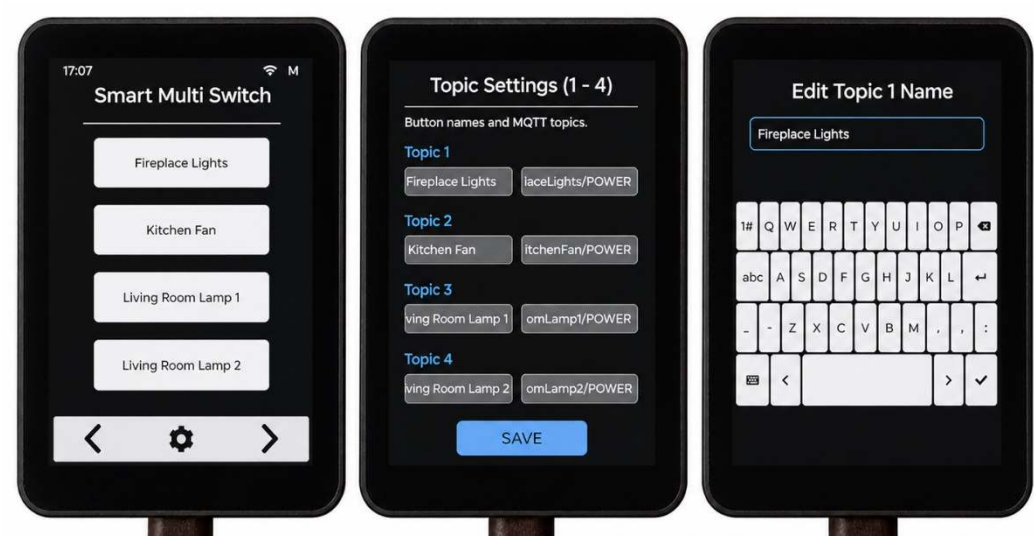


Figure 9.8 – Running application screens

So far, we have focused on building an intuitive and responsive touch-based interface for controlling devices via MQTT. The system now provides a solid foundation for user interaction and configuration using a TFT display.

In the next sections, we will take a step forward by integrating a camera into the application and learning how to create a cheaper camera node for your home.

Integrating cameras with ESPHome and Home Assistant

Introducing ESP32 camera sensors

Before integrating cameras with the touchscreen display or Home Assistant, it is important to understand the camera sensors commonly used with ESP32-S3 camera modules. The most common options are the OV2640, OV3660, and OV5640 from Omnivision (<http://www.ovt.com>). These sensors differ mainly in resolution, image quality, memory requirements, and performance. *Table 9.3* compares some of the common camera sensors available at the time this book is being written.

Camera Sensor	Max Resolution	JPEG Support	Image Quality
OV7670	VGA (640×480)	No	Low

Camera Sensor	Max Resolution	JPEG Support	Image Quality
OV2640	2 MP	Yes	Good
OV3660	3 MP	Yes	Better
OV5640	5 MP	Yes	Best

Table 9.3 – Omnivision sensors comparison

For smart home use cases, the right choice depends on whether the goal is fast snapshots, better image quality, or higher-resolution monitoring. These camera sensors come with a cable attached and lenses with different Field of View (FOV) angles. *Figure 9.9* shows some of these sensors.



Figure 9.9 – Omnivision camera sensors

The Field of View (FOV) defines how wide an area the camera can capture. It is determined mainly by the lens, not the sensor itself, although modules are typically sold with a fixed lens matched to the sensor. A wider FOV captures more area but with less detail per object, while a narrower FOV provides more detail but covers a smaller scene. *Table 9.5* lists typical FOVs and some features associated with them.

FOV Angle	Type	Coverage	Typical Use Case	Pros	Cons
60° – 75°	Narrow	Focused area	Entry door, face capture	Higher detail, less distortion	Limited coverage
75° – 90°	Standard	Balanced view	General indoor monitoring	Good balance of detail and area	Moderate distortion
90° – 120°	Wide	Large area	Garage, living room	Covers more space	Slight edge distortion
120° – 160°	Ultra-wide	Very large area	Corners, ceiling mounts	Maximum coverage	Noticeable distortion (fish-eye effect)

Table 9.5 – Camera FOV angles and features

As you now know more about camera sensors, let's move to the next subsection where we will comment about the approach used in the display application and what camera sensor was chosen for it.

Approach and the camera sensor used for the application

Connecting a camera directly to the Waveshare ESP32-S3 display board is technically possible, but it should be used carefully. The same ESP32-S3 in this board would be responsible for both the graphical interface and camera processing, which increases CPU, memory, and PSRAM usage. That said, we will use ESPHome to stream the camera sensor connected to the display in low resolution, exposing it to Home Assistant rather than directly to the display. In the next section, we will explore more about an independent camera module using a separate ESP32-S3.

For this application, the selected sensor is an OV3660 with a 120-degree FOV. Next, we will see how to configure the camera module in the ESPHome application.

Configuring a camera sensor module in ESPHome

We will use the esp32_camera ESPHome component (https://esphome.io/components/esp32_camera/) to configure the camera module connected to the Waveshare ESP32-S3 display

board. The first code snippet required is the I2C bus and pins configuration and is presented as follows.

```
i2c:
  - id: bus_a
    sda: GPIO8
    scl: GPIO7

esp32_camera:
  id: cam
  name: "Smart Casa Monitor Camera"

external_clock:
  pin: GPIO38
  frequency: 20MHz

i2c_id: bus_a

data_pins:
  - GPIO45
  - GPIO47
  - GPIO48
  - GPIO46
  - GPIO42
  - GPIO40
  - GPIO39
  - GPIO21

vsync_pin: GPIO17
href_pin: GPIO18
pixel_clock_pin: GPIO41
```

ESP32 camera modules use a parallel camera interface called DVP (Digital Video Port) to communicate with image sensors such as OV2640, OV3660, and OV5640. This interface transfers pixel data in parallel using synchronized clock and control signals. Among the pins required for the camera connectivity we have the following:

- I2C bus is used by the ESP32 to configure sensor parameters like resolution, brightness, etc. We need two pins to be configured as Serial Data (*sda*) and Serial Clock (*scl*).

- `data_pins`: it is an 8-bit parallel bus which carries the pixel data from the camera sensor to ESP32. At each clock cycle one byte of image data is transferred.
- **Synchronization and Control Signals**: the `pixel_clock_pin` indicates when each byte of data is valid, `href_pin` indicates valid data for each row and `vsync_pin` signals the start of a new frame.

Other parameters we can configure for the camera are presented in the sequence:

```
max_framerate: 12 fps
idle_framerate: 0.2 fps
resolution: 640x480
jpeg_quality: 12
contrast: 1
brightness: 1
saturation: 0
vertical_flip: true
horizontal_mirror: false
```

These camera parameters can be grouped into three main areas: image quality and performance, image appearance, and orientation. Settings like `resolution`, `jpeg_quality`, and `max_framerate` directly impact the image clarity, quality and smoothness but require more processing and memory. Parameters such as `contrast`, `brightness`, and `saturation` adjust how the image looks by enhancing edges, light levels, and color intensity without affecting performance. Finally, `vertical_flip` and `horizontal_mirror` control the image orientation, ensuring it appears correctly based on how the camera is physically mounted.

Using only the `esp32_camera` component, the camera can already be fully integrated with Home Assistant as a native camera entity, allowing snapshots and streaming directly within dashboards and automations. However, adding the `esp32_camera_web_server` configuration provides an additional, independent way to access the camera through standard HTTP endpoints, without relying on Home Assistant. We can use the following parameters and configuration to do it.

```
esp32_camera_web_server:
- port: 8080
  mode: stream
- port: 8081
  mode: snapshot
```

`esp32_camera_web_server` creates web access points for the ESP32 camera. In this configuration, port 8080 runs in stream mode, which provides a continuous MJPEG video stream that can be opened from a browser or used by Home Assistant. Port 8081 runs in snapshot mode, which returns a single JPEG image each time it is requested. This gives the camera two access methods: one for live monitoring and another for lightweight capture.

The camera can be accessed using the address `http://smart-casa-monitor.local:8080` for streaming and `http://smart-casa-monitor.local:8081` for snapshot. *Figure 9.10* shows how the camera image looks.

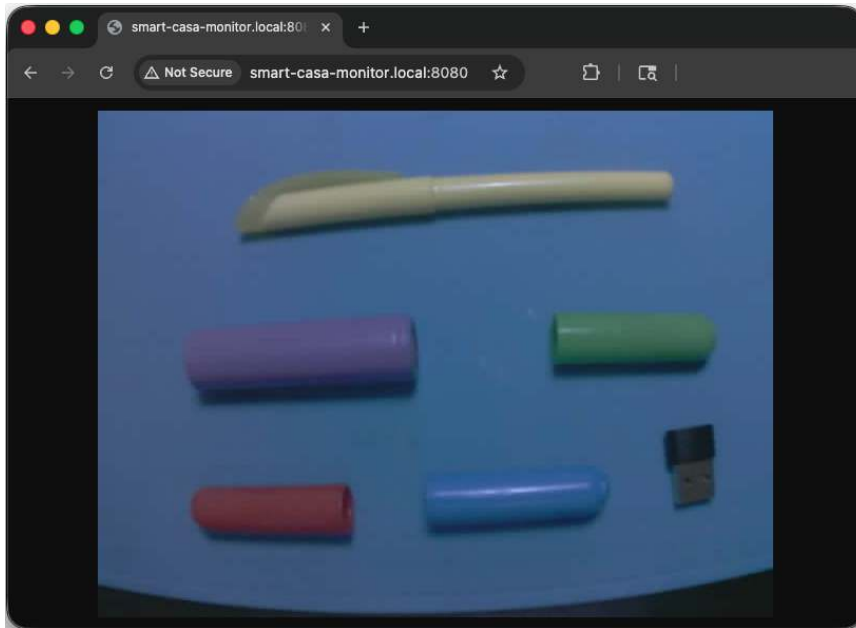


Figure 9.10 – ESPHome camera snapshot

The camera image can also be obtained by searching for the Smart Casa Monitor device in Home Assistant.

Combining a touchscreen display and camera on the same ESP32-S3 offers a compact and versatile solution, but it limits processing and flexibility since one device handles everything. To overcome this, the next section introduces a separate ESP32-S3 device dedicated to the camera, improving performance and scalability as an independent camera.

ESP32-S3 camera as an independent device

In the previous section, the camera module sensor was integrated into the touchscreen display device, combining visualization and image capture into a single unit. This approach is practical for compact solutions but it introduces processing limitations and reduces flexibility. In this section, we shift to a more scalable architecture by using the ESP32-S3 camera module as an independent device.

Hardware architecture

The hardware used to implement these cameras is called the ESP32-S3-CAM board and comes with an OV-3660 sensor. This kit is available in most electronic retailers for around USD 7.

Figure 9.11 shows a picture of this board with and without a 3D-printed enclosure.

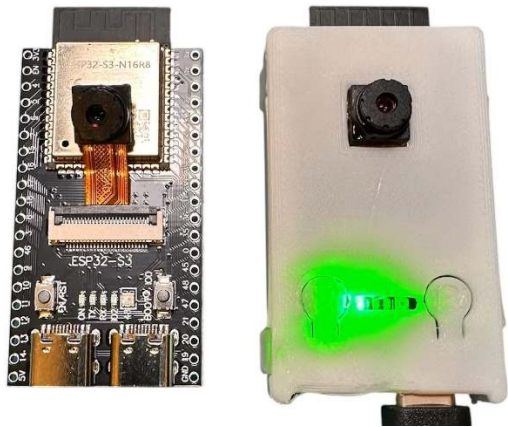


Figure 9.11 – ESP32-S3-CAM board and OV-3660 sensor

Note

The camera's 3D-printed enclosure is available at: <https://www.printables.com/model/1414018-esp32-s3-wroom-casecover>

The ESP32-S3-CAM board has the following features presented in *Table 9.6*.

Feature	Description
CPU	Dual-core Xtensa 32 Bits LX7 (up to 240 MHz) based. ESP32-S3-N16R8 module

Feature	Description
Wireless Communication	2.4 GHz 802.11 b/g/n Wi-Fi, Bluetooth 5, BLE
Camera	DVP Camera Interface. OV2640 / OV3660 /OV5640 Sensor support
Memory	384 KB ROM, 512 KB SRAM, 16 KB RTC SRAM, 8 MB PSRAM, 16MB Flash
Storage	MicroSD card interface
USB	1 x Native USB OTG, 1 x USB/UART Programming

Table 9.6 – ESP32-S3-CAM board features

Software for the ESP32-S3-CAM board was implemented to allow you to connect the camera to Home Assistant using an IP or a local DNS address. This software will be detailed in the next subsection.

Camera software features and implementation

The software developed uses the ESP-IDF 6.0 compiler from Espressif. VS Code was used as the development platform.

The main software features are:

- **HTTP-Based Camera Interface:** the software exposes HTTP endpoints that allow the camera to be accessed and configured.
- **Network independence:** The system avoids reliance on static IP addresses by using mDNS (Multicast DNS).
- **Wi-Fi provisioning via captive portal:** The software includes a self-hosted configuration portal, eliminating the need to hardcode Wi-Fi credentials.

The next subsection will detail each one of these main features implemented.

HTTP-based camera interface feature

The ESP32-S3 camera firmware implements a lightweight HTTP server through the function `start_webserver(void)`, which initializes the web service and registers all available endpoints. These include `/` for the main interface, `/capture` for JPEG snapshots, `/stream` for MJPEG video streaming, `/status` for JSON-based camera information, `/control` for runtime parameter adjustment, and `/reset_sensor` for reinitializing the camera sensor.

The main web page is handled by `index_handler(httpd_req_t *req)`, where the parameter `httpd_req_t *req` represents the incoming HTTP request. This function serves a simple interface that allows users to access camera features directly from a browser. This main page is presented on *Figure 9.12*.

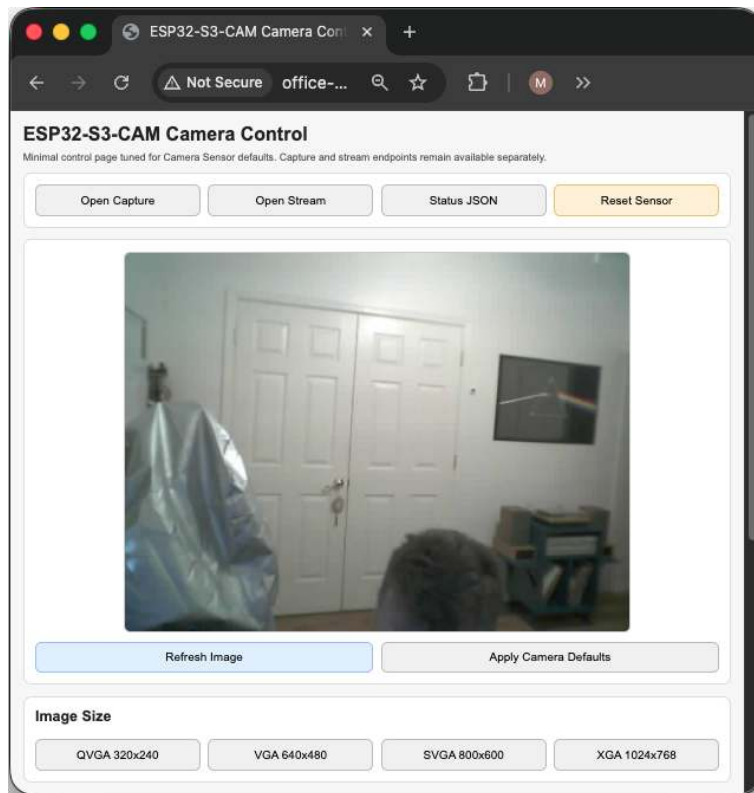


Figure 9.12 – ESP32-S3-CAM main HTTP page

Snapshot functionality is implemented in `capture_handler(httpd_req_t *req)`, which captures a single frame and returns it as a JPEG image. A complementary function, `capture_head_handler(httpd_req_t *req)`, processes HTTP HEAD requests, allowing clients to verify endpoint availability without downloading image data.

Live video streaming is handled by `stream_handler(httpd_req_t *req)`, which continuously captures frames and sends them as an MJPEG stream. This endpoint is particularly important for integration with Home Assistant.

The function `status_handler(httpd_req_t *req)` provides a JSON response with the current camera configuration, including parameters such as resolution, brightness, contrast, and

orientation. Runtime configuration is performed through `control_handler(httpd_req_t *req)`, which parses query parameters like `/control?var=brightness&val=1` and applies them directly to the sensor.

Finally, `reset_sensor_handler(httpd_req_t *req)` allows the camera sensor to be reset via HTTP, restoring its default configuration without restarting the device.

Network independence with mDNS

To eliminate dependence on hardcoded static IP addresses, the system uses Multicast DNS (mDNS) through the function `mdns_init_camera(void)`. This function initializes the mDNS service and registers a hostname such as `office-camera.local`, allowing the device to be accessed using a human-readable name instead of a dynamically assigned IP address. If you need to change the host name, just change the `define MDNS_HOSTNAME`.

With this configuration, users can access the camera using URLs like `http://office-camera.local/`, `http://office-camera.local/capture`, and `http://office-camera.local/stream`. This significantly improves usability and reliability in DHCP-based networks, where IP addresses may change over time.

Wi-Fi provisioning via captive portal

The firmware includes a self-hosted Wi-Fi provisioning system that removes the need to hardcode credentials. The process begins with `init_nvs(void)`, which initializes non-volatile storage (NVS) so that configuration data can be saved persistently.

Stored credentials are accessed using `load_wifi_credentials(char *ssid, size_t ssid_len, char *pass, size_t pass_len)`, which reads the SSID and password from NVS and returns a boolean indicating whether valid data exists. Credentials are saved using `save_wifi_credentials(const char *ssid, const char *pass)`, ensuring that the device can reconnect automatically after reboot.

If no credentials are found, the function `start_setup_portal(void)` starts the ESP32 in Access Point mode, creating a network such as `ESP32-CAM-SETUP` with the configuration page available at `http://192.168.4.1`. The setup interface is served by `setup_page_handler(httpd_req_t *req)`, which displays the form presented on *Figure 9.13* for entering Wi-Fi credentials.

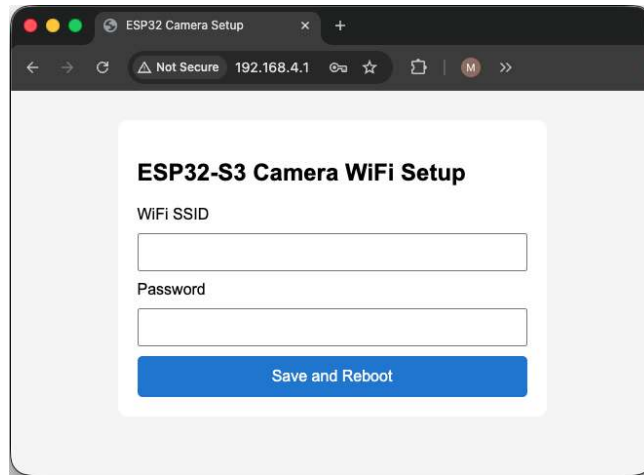


Figure 9.13 – ESP32-S3-CAM Wi-Fi credentials

When the user submits the form, `save_handler(httpd_req_t *req)` processes the input, extracts the SSID and password, stores them in NVS, and restarts the device to apply the new configuration.

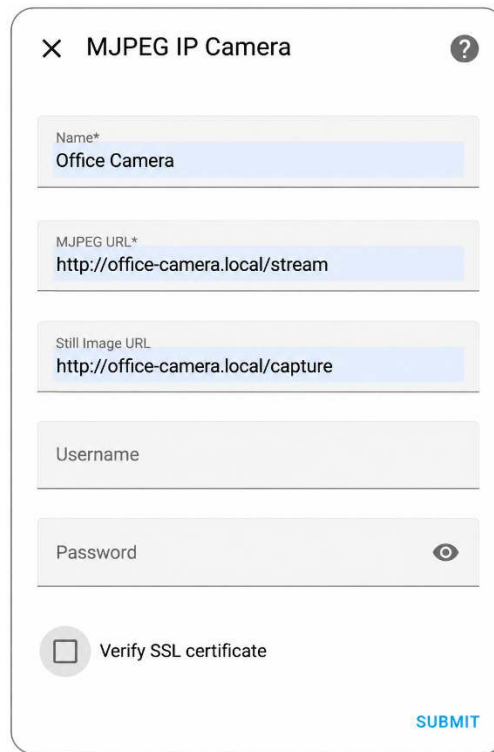
Once credentials are available, `wifi_init_sta_from_nvs(void)` connects the ESP32 to the configured network. The connection process is monitored by `wifi_event_handler(void *arg, esp_event_base_t event_base, int32_t event_id, void *event_data)`, which handles Wi-Fi and IP events such as connection, disconnection, and IP assignment, ensuring reliable network operation.

After detailing the main features and the software function implementation, we will explore in the next subsection how to connect the camera to Home Assistant.

Integrating the ESP32-S3 camera with Home Assistant

The ESP32-S3-CAM can be easily integrated into Home Assistant using the MJPEG IP Camera integration. Since the firmware exposes an MJPEG stream through the `/stream` endpoint and snapshots through `/capture`, no additional protocols such as RTSP are required. I created *Video 9.2* (<https://packt.link/coFJ0>) to show how to connect the camera to Home Assistant.

To add the camera, using Home Assistant navigate to Settings | Devices & Services | Add Integration | MJPEG IP Camera. Then configure the camera with the parameters presented in *Figure 9.14* and click on SUBMIT.



The screenshot shows a configuration form titled 'MJPEG IP Camera' with a close button (X) and a help icon (?). The form contains several input fields: 'Name*' with the value 'Office Camera', 'MJPEG URL*' with the value 'http://office-camera.local/stream', and 'Still Image URL' with the value 'http://office-camera.local/capture'. Below these are empty fields for 'Username' and 'Password'. At the bottom, there is a checkbox labeled 'Verify SSL certificate' which is currently unchecked. A blue 'SUBMIT' button is located in the bottom right corner.

Figure 9.14 – ESP32-S3-CAM configuration in Home Assistant

In the next screen, just add the home area where you want the camera to be installed, and that's it. Your camera is now integrated into Home Assistant, and you can see it as part of the Home Assistant devices.

Integrating SD card stream recording is something you might want to consider. As the ESP32-S3-CAM board has an SD-card interface, this is an additional feature that can be incorporated into this application.

The ESP32-S3 camera system developed in this section demonstrates how a low-cost platform can become a fully functional smart camera. By combining an HTTP interface, mDNS for network independence, and a captive portal for Wi-Fi setup, the device evolves into a practical and scalable solution to implement in your home.

Summary

Chapter 9 presents a hands-on project that combines ESPHome, MQTT, and ESP32 hardware to build an interactive smart home device with a touchscreen and camera. The Smart Casa Monitor enables direct user interaction in a useful application that includes eight MQTT-enabled switches controlled by a single device.

The chapter covers display and touchscreen technologies, followed by hardware integration using an ESP32-S3 based board. It then guides the creation of a touchscreen UI using ESPHome and LVGL, including navigation, MQTT communication, and configurable controls.

It also introduces camera integration, showing how to stream video to Home Assistant and comparing integrated versus standalone ESP32 camera solutions. The independent camera approach adds scalability with HTTP streaming, mDNS, and Wi-Fi provisioning.

In *Chapter 10, Advanced Automation and External Logic*, you will explore advanced automation design, including state machines, conditional logic, and external tools like Node-RED and AppDaemon to build robust and smart home systems.

Part 4

Advanced Automation Patterns and AI Integration

In this final part, you'll explore advanced automation logic using AppDaemon, Node-RED, HACS, and AI-based techniques. By the end of this part, you'll be able to design more sophisticated automations, extend Home Assistant capabilities, and use AI to detect patterns and suggest optimizations.

This part of the book includes the following chapters:

- *Chapter 10, Advanced Automation and External Logic*
- *Chapter 11, Extending Home Assistant with HACS*
- *Chapter 12, AI-Driven Home Automation*

10

Advanced Automation and External Logic

Home Assistant provides an automation engine capable of handling a wide range of smart home scenarios using triggers, conditions, and actions. As your automation system grows, however, simple automations can become difficult to maintain and scale. Complex workflows often require advanced logic structures, external scripting tools, and better strategies to handle failures and unexpected behaviors. Understanding concepts such as state machines, sequences, and conditional flows allows you to create smarter, more reliable, and maintainable automation systems.

Building on the previous chapters, where you explored advanced Home Assistant configurations, communication protocols, and custom devices, this chapter focuses on designing automation systems that behave like professional software applications. You will learn how to extend Home Assistant using AppDaemon for Python-based automations and Node-RED for visual workflow creation and complex integrations. You will also learn how to choose the right tool for different automation challenges and design resilient systems with key principles.

By the end of this chapter, you will be able to design complex automation patterns, develop Python-based automations with AppDaemon, create advanced Node-RED workflows, choose the most appropriate automation approach for different scenarios, and build more resilient home automation systems.

In this chapter, we will explore the following content:

- Beyond basic automations: state machines, sequences, and conditional flows
- AppDaemon: Python-based automation and custom applications
- Node-RED: visual programming for complex integrations and flows

- Orchestrating external automation logic: choosing the right tool
- Designing resilient home automation systems

Technical requirements

To follow the examples in this chapter, you should have the following:

- A working Home Assistant installation using either Home Assistant OS or Home Assistant Container
- Advanced Mode enabled in Home Assistant
- Basic experience creating native Home Assistant automations
- Familiarity with editing YAML files
- Familiarity with the Home Assistant Developer Tools
- Basic Python programming knowledge to understand the AppDaemon examples
- Basic understanding of logic flows to follow the Node-RED examples
- Node-RED app installed in Home Assistant
- AppDaemon app installed in Home Assistant
- Visual Studio Code or another code editor installed
- MQTT Broker configured in Home Assistant
- At least one smart device or virtual entity for automation testing
- Basic experience with YAML snippets, Python scripts, and Node-RED flows

This chapter assumes you already have the foundational Home Assistant environment configured and operational from the previous chapters.

Beyond basic automations: state machines, sequences, and conditional flows

Home Assistant automations are one of the most powerful features of the platform. A simple automation can react to a sensor, check a condition, and execute an action. This is enough for many use cases, such as turning on a light when motion is detected or sending a notification when a door opens.

However, as a smart home grows, automations often become more complex. A single room may have several sensors, multiple lighting scenes, different behaviors for day and night, and exceptions for guests, sleeping hours, or manual overrides. At this point, writing many small independent automations can become difficult to maintain.

This section introduces automation design patterns that help you move from isolated rules to structured automation systems. You will learn how to think in terms of states, sequences, and conditional flows, so your automations become easier to understand, debug, and expand.

Figure 10.1 compares a basic automation with an advanced automation pattern to give you an idea of what this section will discuss.

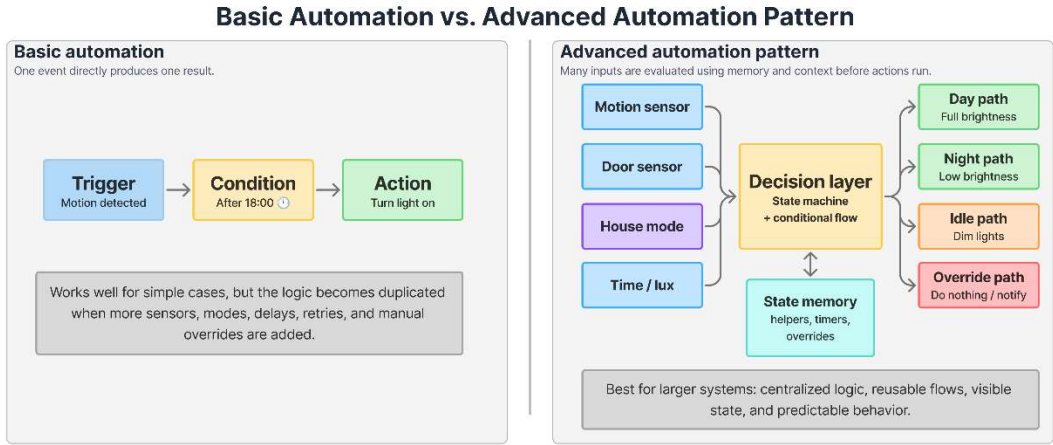


Figure 10.1 – Basic automation x advanced automation pattern

Thinking in systems instead of individual rules

A smart home goes beyond a collection of devices. It is a system where sensors, helpers, modes, user actions, and automations interact. When you design automation logic as a system, each automation has a clear role. For example, a motion sensor should not necessarily decide whether the lights turn on. Its job is only to report motion. Another layer can interpret that motion together with time, room mode, brightness level, and occupancy state. This separation makes the system easier to modify later.

A useful way to organize automation logic is to divide it into three layers: sensing, decision-making, and action. *Figure 10.2* breaks down these layers.

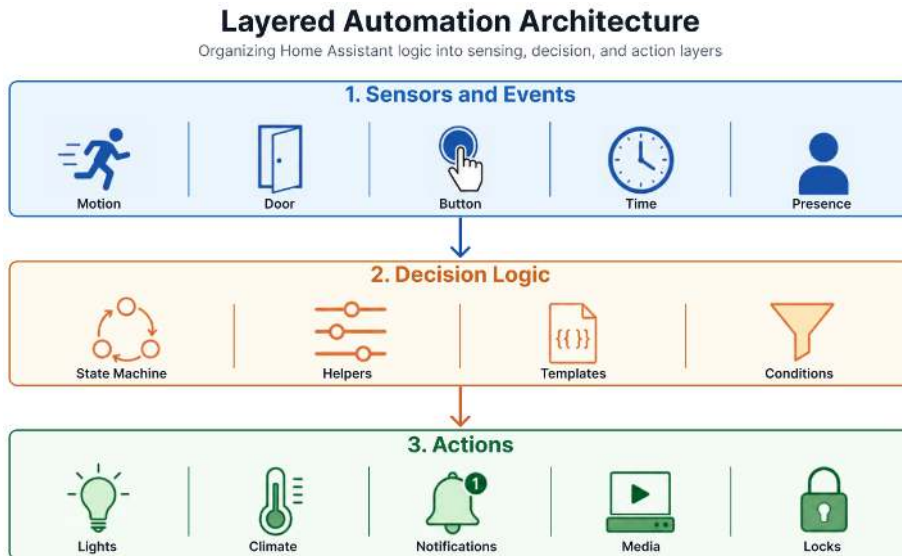


Figure 10.2 – Layered automation architecture

As presented in *Figure 10.2*, the sensing layer receives events from motion sensors, door sensors, buttons, and other inputs. The decision layer evaluates the current context and decides what state the system should be in. The action layer performs the final operation, such as turning on lights, activating a scene, or sending a notification.

Once we start thinking of automations as systems instead of individual rules, state machines give us a practical way to define the states and transitions that control system behavior, as we will see in the next section.

State machines in home automation

The concept of state machines comes from computer science and control systems, where systems are modeled as a set of defined states and transitions between them. A state describes the current situation. A transition describes what causes the system to move from one state to another.

Instead of creating many automations that independently react to events, a state machine gives you one central model of the current situation. This makes the automation logic more predictable and easier to debug.

The state machines concept is very useful in home automation because many behaviors are state-based. For example, a room can be empty, occupied, or idle as presented in *Figure 10.3*.

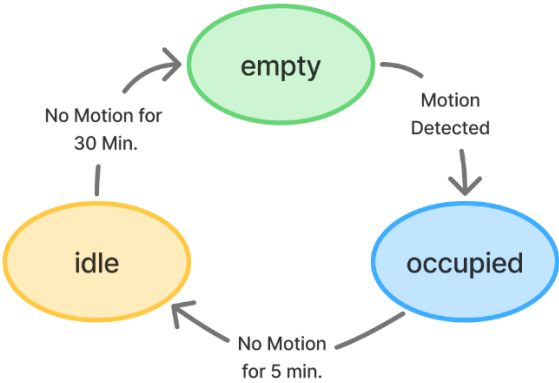


Figure 10.3 – Room occupancy state machine

According to *Figure 10.3*, the room starts as empty. When motion is detected, the state changes to occupied. If no motion is detected for five minutes, the state changes to idle. If no motion is detected for thirty minutes, the state changes to empty.

The idle state is important because it provides a buffer between the occupied and empty states. During this state, the system may dim the lights instead of turning them off immediately.

To create the states for a room we can use helpers. The `input_select` is a practical helper because it allows you to define a list of possible states. It can be created using `Devices & services | Helpers | + Create Helper | Dropdown`. This helper becomes the memory of the room. Other automations can read it to decide what to do.

As an example, let's implement the state machine for the living room. Create the `input_select` helper for it. When a motion is detected in the living room, the room becomes occupied. *Table 10.1* summarizes the states to be implemented.

State Transition	Trigger	Condition	Action
Empty → Occupied	Motion turns on	None	Set <code>input_select.living_room_state</code> to occupied
Occupied → Idle	Motion off for 5 minutes	State is occupied	Set state to idle

State Transition	Trigger	Condition	Action
Idle → Empty	Motion off for 30 minutes	State is idle	Set state to empty

Table 10.1 – State machine automation transition

A pseudo code for the automation that implements the transition from empty to occupied is the following:

```
automation:
  alias: Living Room State Machine - Occupied
  triggers:
    - trigger: state
      entity_id:
        - binary_sensor.motion
      to: "on"
  actions:
    - action: input_select.select_option
      target:
        entity_id: input_select.living_room_state
      data:
        option: occupied
```

The automation for the transitions from occupied to idle and idle to empty follow the same format.

With this pattern, the motion sensor only updates the room state. A separate automation can react to the state and control lighting. That is what we will see in the next subsection.

Separating state logic from action logic

For larger systems, it is easier to split the automation into two parts. The first part identifies the current state, such as whether the room is occupied, idle, or empty. The second part decides what to do with that state, such as turning on a light, dimming it, activating a scene, or turning everything off.

Using the example from the last subsection, the state machine decides that the living room is now occupied. A second automation decides what lighting scene to activate based on time of day, brightness, and house mode.

We can create an automation to deal with the action logic of the Living room occupancy state. This automation is represented using the diagram in *Figure 10.4*.

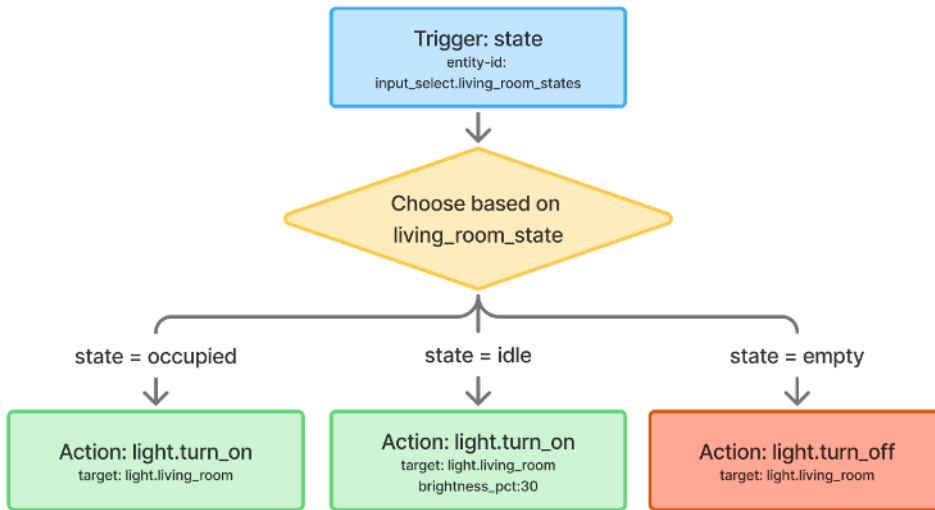


Figure 10.4 – Living room lighting based on state

Figure 10.4 shows a conditional flow allowing the automation to branch depending on context. This is possible because Home Assistant's choose action which works like an if / elif / else structure in programming. It evaluates a list of conditions and executes the first matching sequence.

When actions involve the execution of different steps, there are some ways to understand and handle them. We will see this in the next subsection.

Sequences and multi-step automations

A sequence is an ordered list of actions. Home Assistant executes each action in order, from top to bottom. Sequences are useful when the automation needs to perform a workflow rather than a single action.

For example, a hallway light may turn on, remain active for ten seconds, and then turn off as the example below.

```

actions:
- action: light.turn_on
  metadata: {}
  data: {}
  target:
    device_id: your_hallway_light_id
  
```

```
- delay:
  hours: 0
  minutes: 0
  seconds: 10
  milliseconds: 0
- action: light.turn_off
  metadata: {}
  data: {}
```

This is simple, but sequences can be much more powerful. You can combine service calls, delays, conditions, notifications, scripts, and waits. A good example is a "*leaving home*" sequence that turns off lights, lowers the thermostat, checks whether doors are closed, and arms the alarm.

Sequences help you create predictable workflows. They also make automations easier to read because each step is listed explicitly.

If a sequence is used in more than one automation, it is often better to move it into a script. Scripts make automation logic reusable. Instead of duplicating the same action list in several places, you define it once and call it from different automations. We saw an example of script in Chapter 5 in the "*Hands-on Project: Integrating Wi-Fi and Zigbee for Garage Gate Control*" section.

When the sequence includes a choose statement, a best practice is to include a default path. This is useful when none of the listed conditions match. A default path prevents the automation from silently doing nothing. It typically can turn on a safe fallback scene, send a notification, or create a log entry. In advanced systems, this is useful for troubleshooting because it shows when the automation reaches an unexpected condition.

You can combine state machines with conditional flows to create a complete set of actions related to a particular room in your home. We will see this in the next subsection.

Combining state machines and conditional flows

State machines and conditional flows become powerful when used together. The state machine stores the current situation, while conditional flows decide what action to perform for that situation.

For example, the living room state machine may detect that the room is occupied. After that, the automation can check what activity is happening in the room, such as watching TV, reading, relaxing, or normal use. Based on that activity, it can choose the correct lighting behavior. *Figure 10.5* shows this second step, where the room state is combined with the selected activity to decide what action should run.

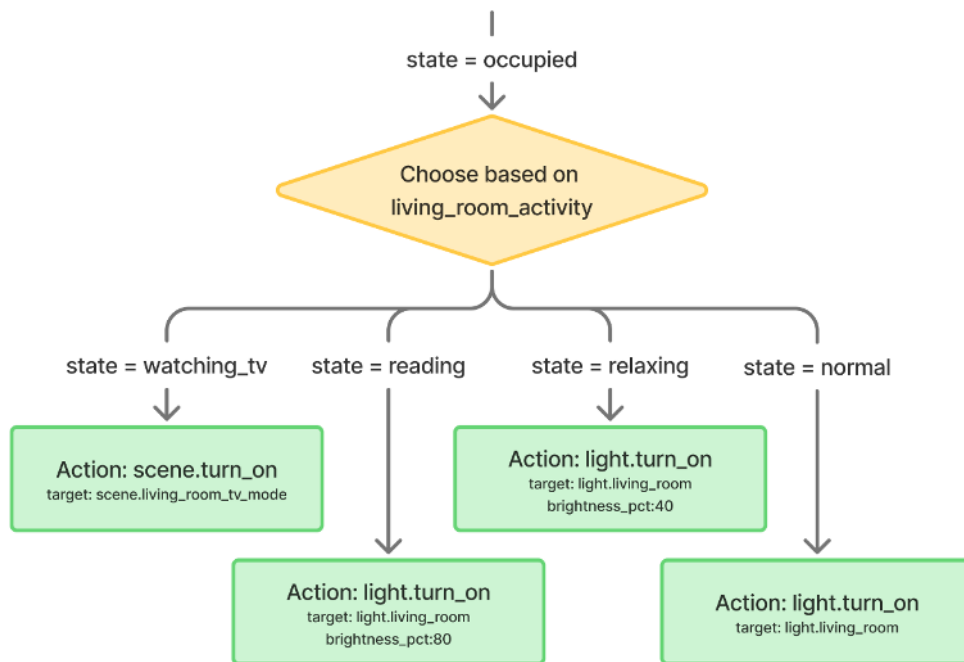


Figure 10.5 – Mixing state machine and conditional flows example

The automation diagram example uses an `input_select.living_room_state` as the state machine memory and `input_select.living_room_activity` as the context selector. The state machine determines whether the room is occupied, idle, or empty, while the choose block decides which lighting behavior should run for each situation. The YAML code for this automation is in the book repository (Living Room Lighting - State and Activity).

When state machines and conditional flows are combined, the automation logic becomes more powerful, but it also becomes easier to spread decisions across too many helpers, scripts, and automations. To avoid this, name automations clearly and document their role. Use helpers with descriptive names. Keep state transitions centralized. Avoid having several automations update the same helper unless each transition is obvious and intentional.

A good naming convention can make a large automation system easier to understand:

```
Living Room State Machine - Occupied
Living Room State Machine - Idle
Living Room State Machine - Empty
Living Room Lighting - State and Activity
Living Room Lighting - Manual Override
```

Basic automations are useful, but they become harder to maintain as devices, conditions, and exceptions increase. State machines, sequences, and conditional flows help manage this complexity by defining clear states, organizing ordered actions, and choosing the right behavior based on context.

With these concepts in place, the next step is to explore how external automation tools can extend Home Assistant even further, starting with AppDaemon.

AppDaemon: Python-based automation and custom applications

Home Assistant's native automation engine is powerful and flexible, especially when combined with YAML, helpers, scripts, scenes, and templates. However, as automations grow in complexity, some logic can become difficult to maintain using only YAML. Long conditional blocks, repeated logic, timers, loops, and multi-step decision flows may become hard to read and debug. This is where AppDaemon becomes useful.

AppDaemon is a Python-based automation engine that connects to Home Assistant and allows you to write automations as Python applications.

With AppDaemon you can create reusable Python classes, define helper methods, store internal variables, manage timers, and organize your automations as small applications. This makes it a good option when your automation logic starts to look more like a program than a configuration file.

Figure 10.6 shows a comparison between native Home Assistant automations and AppDaemon.

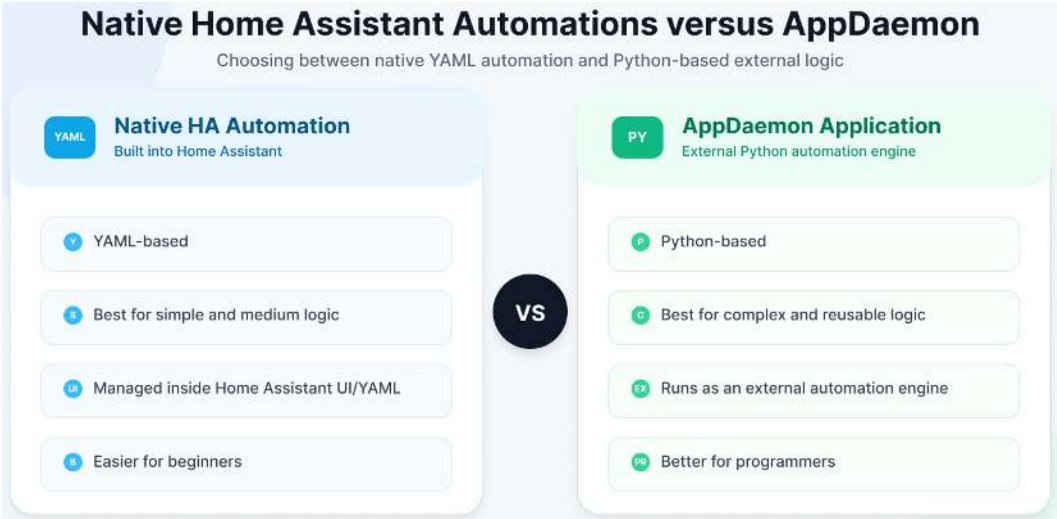


Figure 10.6 – Native Home Assistant automations versus AppDaemon

What is AppDaemon?

AppDaemon is an external automation framework that communicates with Home Assistant through its API. It listens for state changes, events, and service calls, then executes Python code based on the logic defined by the user. AppDaemon also includes a configurable dashboard system called HADashboard, which can be used to create wall-mounted control panels for tablets and touchscreens. More information about AppDaemon can be found at: <https://AppDaemon.readthedocs.io/>.

The AppDaemon Hass plugin connects to Home Assistant using the WebSocket API and maintains the connection while AppDaemon is running. It also uses the REST API for some operations.

Figure 10.7 shows how AppDaemon is connected to Home Assistant.



Figure 10.7 – AppDaemon connection with Home Assistant diagram

In practical terms, AppDaemon allows you to write automations using Python instead of YAML. Each automation is usually implemented as a Python class. When AppDaemon starts, it loads the Python apps, runs their `initialize()` methods, and begins listening for events or state changes.

This approach is useful because Python gives you access to variables, loops, functions, classes, dictionaries, lists, external libraries, and structured error handling. If you are familiar with Python, these features allow you to build automation logic that is easier to reuse and expand over time.

Installing AppDaemon

AppDaemon can be installed in different ways, including Docker and Python environments. In a Home Assistant OS installation, the most convenient approach is usually to install it as an app. The Home Assistant Community AppDaemon app provides AppDaemon as a packaged service for Home Assistant users.

The app installation process in Home Assistant follows the standard process for Apps installations. It is preconfigured to automatically connect to Home Assistant. No extra configuration is required.

The exact folder structure may vary depending on the app configuration, but AppDaemon apps are commonly stored under a path similar to `/addon_configs/a0d7b954_AppDaemon`.

In the AppDaemon app configuration page in Home Assistant, you will find options to configure system packages, Python packages and init commands. The system and Python package options allow AppDaemon to load specified packages when AppDaemon starts. The init commands allow some shell commands to be executed before AppDaemon starts. These fields are presented in *Figure 10.8*.

← Info Documentation **Configuration** Log

AppDaemon

Options

System packages*

Allows you to specify a list of additional Alpine Linux packages to be installed before AppDaemon starts.

Python packages*

Allows you to specify a list of additional Python packages to be installed before AppDaemon starts.

Init commands*

Allows you to specify a list of custom shell commands that are ran before AppDaemon starts.

☐ Show unused optional configuration options

Save

Figure 10.8 – AppDaemon Home Assistant app configurations

Now that AppDaemon is installed, let's see in the next section how a basic AppDaemon application works.

Basic AppDaemon application structure

An AppDaemon app is usually created as a Python class that inherits from the Home Assistant AppDaemon API class. The `initialize()` method is executed when the app starts. This is where you normally register listeners, initialize variables, and prepare the app logic.

The following example listens to a motion sensor. When the sensor changes to on, the app turns on a living room light and then turns the light off after a delay of 5 minutes.

```
import appDaemon.plugins.hass.hassapi as hass

class MotionLightTimer(hass.Hass):

    def initialize(self):
        self.handle = None
```

```
self.listen_state(self.motion_detected, "binary_sensor.motion")

def motion_detected(self, entity, attribute, old, new, kwargs):
    if new == "on":
        self.turn_on("light.living_room")

        if self.handle is not None:
            self.cancel_timer(self.handle)

        self.handle = self.run_in(self.turn_off_light, 300)

def turn_off_light(self, kwargs):
    self.turn_off("light.living_room")
    self.handle = None
```

In this example, `listen_state()` is an AppDaemon API method that registers a listener for changes in `binary_sensor.motion`. It is not directly calling the Home Assistant REST API to read the state. AppDaemon maintains its connection with Home Assistant and receives state updates, mainly through the WebSocket API. When `binary_sensor.motion` changes state, AppDaemon calls the registered callback method, `motion_detected()`. The method then checks whether the new state is on and calls `turn_on()` to turn on `light.living_room`.

The app uses a timer handle stored in `self.handle`. If a previous timer already exists, it is canceled. Then a new timer is created using `run_in()`. After 300 seconds, the `turn_off_light()` method is executed.

The AppDaemon API provides callback and high-level functions for listening to state, events and log changes, scheduling tasks, manipulating entities, and calling actions.

You will have to create the Python application file (.py) inside the path `/addon_configs/a0d7b954_AppDaemon/apps`. If you want to use the VS Code editor Home Assistant app, you will have to add a symbolic link to the AppDaemon folder location using the command:

```
ln -s /addon_configs/a0d7b954_appdaemon /config/appdaemon_addon
```

Creating the Python file is only part of the process. AppDaemon also needs to know which apps to load. This is commonly done using an `apps.yaml` file. For the example we are using now,

here is the content that has to be added to the apps.yaml file located in the same folder where the Python file for the application was created.

```
motion_light_timer:  
  module: motion_light_timer  
  class: MotionLightTimer
```

The `motion_light_timer` key is the app name. The `module` value refers to the Python file name without the `.py` extension. The `class` value refers to the Python class inside the file.

After creating the Python file and configuring the apps.yaml file, the AppDaemon app needs to be restarted to run the automation.

In summary, the following steps are required to create and run the `motion_light_timer` automation using AppDaemon:

1. Confirm that AppDaemon is already installed, started, configured, and accessible through VS Code or another file editor
2. Create the Python app file, in the case of the example, `motion_light_timer.py`
3. Add the Python code to the file
4. Register the app in apps.yaml
5. Restart AppDaemon
6. Check the AppDaemon logs. You should look for a message like `INFO AppDaemon: Calling initialize() for motion_light_timer` and `INFO AppDaemon: App initialization complete` with no warning or error messages.
7. Test the automation

If everything goes well, you have your first automation implemented using AppDaemon.

Making the app reusable

A major advantage of AppDaemon is reusability. Instead of hardcoding the motion sensor and light entity inside the Python file, you can pass them as parameters from apps.yaml.

Let's see the example of the motion light timer adapted to be reusable in the living room and the kitchen. The `apps.yaml` will have to be modified to be the following:

```
living_room_motion_light:  
  module: motion_light_timer  
  class: MotionLightTimer  
  motion_sensor: binary_sensor.living_room_motion  
  target_light: light.living_room
```

```
    timeout: 300

kitchen_motion_light:
    module: motion_light_timer
    class: MotionLightTimer
    motion_sensor: binary_sensor.kitchen_motion
    target_light: light.kitchen
    timeout: 180
```

The Python app can then read these values using `self.args`. The code below shows how this can be done:

```
import appDaemon.plugins.hass.hassapi as hass

class MotionLightTimer(hass.Hass):

    def initialize(self):
        self.handle = None
        self.motion_sensor = self.args["motion_sensor"]
        self.target_light = self.args["target_light"]
        self.timeout = self.args.get("timeout", 300)

        self.listen_state(self.motion_detected, self.motion_sensor)

    def motion_detected(self, entity, attribute, old, new, kwargs):
        if new == "on":
            self.turn_on(self.target_light)

            if self.handle is not None:
                self.cancel_timer(self.handle)

            self.handle = self.run_in(self.turn_off_light, self.timeout)

    def turn_off_light(self, kwargs):
        self.turn_off(self.target_light)
        self.handle = None
```

With this approach, the same Python class can control different rooms and only the configuration changes. This is one of the most important reasons to use AppDaemon when building larger automation systems that require replicating same functionality among different devices and entities.

HADashboard in AppDaemon

In addition to running Python-based automations, AppDaemon also includes HADashboard, a dashboard system designed for Home Assistant. HADashboard is especially useful for creating simple control panels for wall-mounted tablets, touchscreens, or fixed displays around the house.

Unlike the normal Home Assistant dashboard, HADashboard is configured using YAML files. This makes it a good fit for users who already prefer file-based configuration and want a lightweight interface dedicated to specific devices, rooms, or functions.

HADashboard dashboards are stored in a dashboards directory. The dashboard files use the .dash extension, and each file defines widgets and a layout.

Enabling and creating a basic HADashboard

After AppDaemon is installed and running, HADashboard can be enabled by adding a hadashboard section to the AppDaemon.yaml1 file. By default, AppDaemon looks for dashboards in a folder named dashboards under the AppDaemon configuration directory.

We will create a dashboard to show the four devices we saw in the previous example for the reusable code, meaning the binary_sensor.living_room_motion, light.living_room, binary_sensor.kitchen_motion, light.kitchen. You will have to follow these steps to do it:

1. Create a file called:

```
/config/appdaemon_addon/dashboards/motion_lights.dash
```

2. Add the following content to it:

```
title: Motion Light Dashboard
widget_dimensions: [120, 120]
widget_margins: [6, 6]
columns: 4

living_room_motion:
  widget_type: binary_sensor
  title: Living Room Motion
  entity: binary_sensor.living_room_motion

living_room_light:
  widget_type: light
  title: Living Room Light
  entity: light.living_room
```

```
kitchen_motion:
  widget_type: binary_sensor
  title: Kitchen Motion
  entity: binary_sensor.kitchen_motion

kitchen_light:
  widget_type: light
  title: Kitchen Light
  entity: light.kitchen

layout:
  - living_room_motion(2x1), living_room_light(2x1)
  - kitchen_motion(2x1), kitchen_light(2x1)
```

3. When AppDaemon is running, open the dashboard in a browser using:

```
http://homeassistant.local:5050/motion_lights
```

Or

```
http://<home_assistant_ip>:5050/motion_lights
```

You will see the widgets shown in *Figure 10.9*.

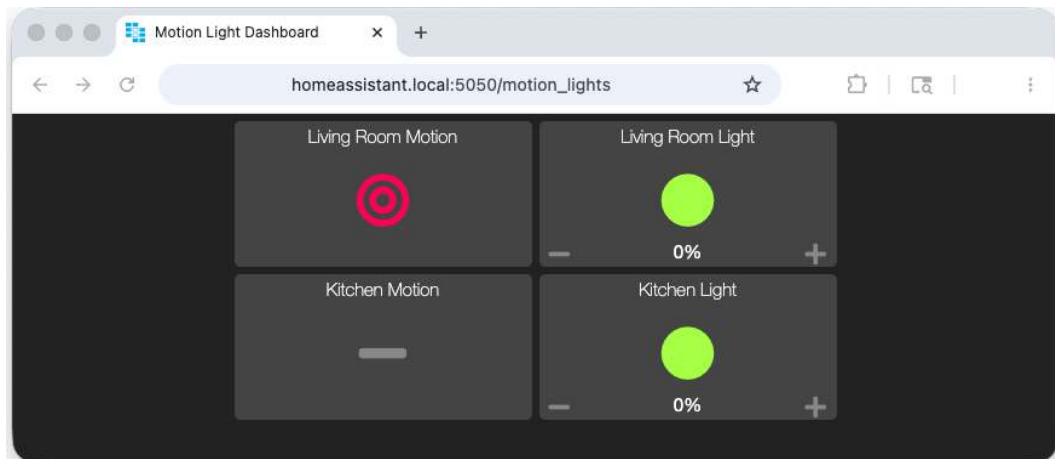


Figure 10.9 – HADashboard example

In *Figure 10.9* you see that the living room motion sensor is activated, which automatically activates the living room light. You can also see the kitchen light on and in this case was activated manually.

AppDaemon extends Home Assistant automations using Python, and it also includes the HADashboard feature for creating custom dashboards to control and monitor entities, states, and automation results.

Native Home Assistant automations are still the best choice for simple workflows. AppDaemon should be used when the logic needs the structure and flexibility of a real programming language. In the next section, we will explore Node-RED, a visual flow-based tool that is useful for complex integrations, data transformations, and automation flows.

Node-RED: visual programming for complex integrations and flows

Node-RED is a flow-based visual programming tool that allows you to create automations by connecting nodes. These nodes are connected together to define how data moves from one step to another. In the context of Home Assistant, Node-RED is especially useful when an automation involves several devices, multiple conditions, delays, transformations, or external services.

Each automation in Node-RED is represented as a flow, and each step in the automation is represented by a node. This makes Node-RED an option for developers who prefer to see the automation logic as a diagram. It also helps when debugging complex flows because each message can be inspected as it moves through the system.

What is Node-RED?

Node-RED is a visual programming environment originally designed for wiring together hardware devices, APIs, and online services. It uses a browser-based editor where users drag nodes into a workspace and connect them to create a flow.

In Home Assistant, Node-RED is commonly used as an external automation engine. It connects to Home Assistant, listens for events or state changes, evaluates logic, and calls Home Assistant services such as turning on lights, sending notifications, or changing climate settings.

For example, the same AppDaemon automation we implemented before for a motion-controlled light can be presented as a simple Node-RED flow in *Figure 10.10*.

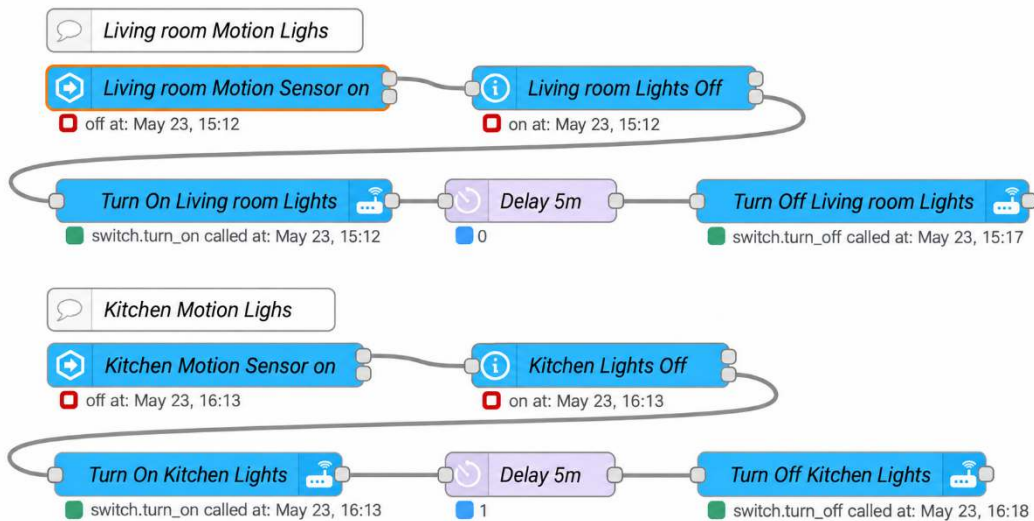


Figure 10.10 – Node-RED motion lights automation example

As you can see in *Figure 10.10*, the execution flow happens from the left to the right following the links between each node, represented by the blue and purple rectangles. There are two execution flows that run independently of each other: one for the living room and the other for the kitchen motion lights.

Figure 10.10 also shows the status when the automation is triggered, with on and off states from the motion sensors and when the living room and kitchen switches are turned on and off. This visual structure makes it easier to understand what happens first, what conditions are checked, and what action is executed at the end.

Installing and using Node-RED

Using Home Assistant OS installation, Node-RED can be installed as a Home Assistant app. Once installed, it runs as a separate service but communicates with Home Assistant through the Home Assistant WebSocket API.

After installation, the Node-RED editor provides a workspace where flows can be created, imported, exported, and organized into tabs. A Node-RED screenshot is presented on *Figure 10.11*.

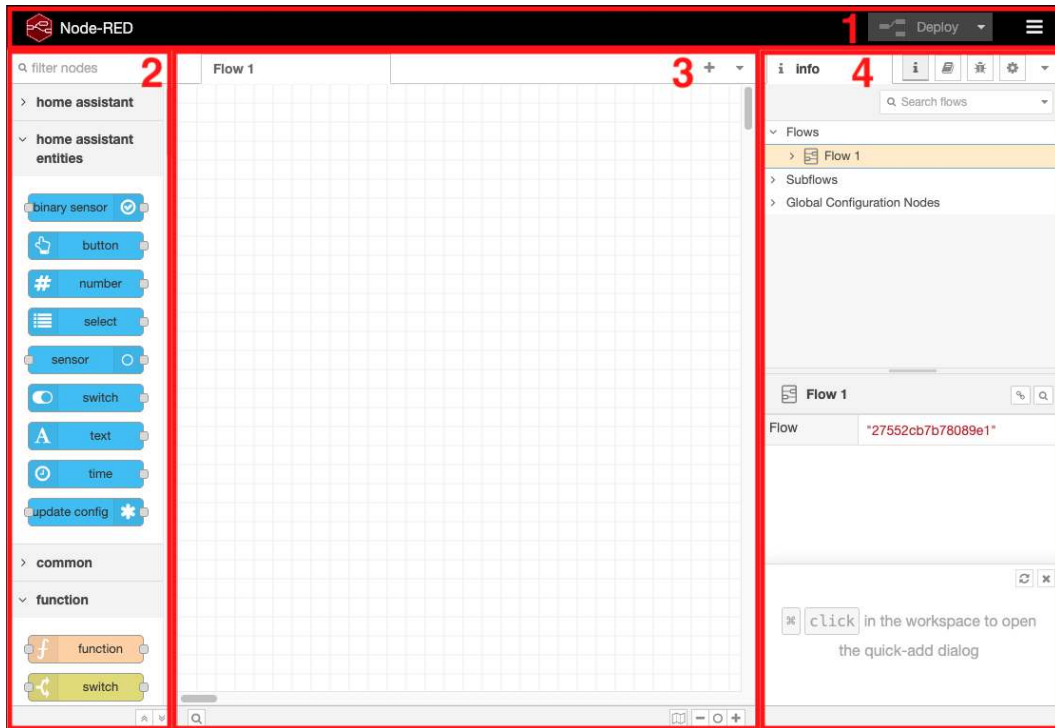


Figure 10.11 – Node-RED screenshot

The red rectangles in *Figure 10.11* can be named as follows:

1. **Header:** Access to the Deploy Button and Main Menu
2. **Palette:** Access to the nodes separated by application
3. **Workspace:** Where the flows will be created. Multiple flows can be created and accessible using the tabs at the top
4. **Sidebar:** Shows flows configuration, debug messages and output information.

Video 10.1 (<https://packt.link/5LZKs>) shows how the automation presented on *Figure 10.10* was created and also how the status messages are presented in the workspace.

Now that you know how to install Node-RED and have an idea on how to use it, you will see in the next steps the main application groups and nodes that can be used to create automation flows.

Key groups and node types

Node-RED includes many types of nodes, organized into application groups according to their purpose. These groups help you quickly find the right node for a specific task, such as receiving data, processing messages, routing logic, calling services, storing values, or debugging a flow.

The exact number of available node groups and nodes can vary depending on the Node-RED installation. A basic installation includes the core palette, but additional nodes can be added through contributed packages. In a Home Assistant environment, extra nodes are installed to support Home Assistant events, services, states, dashboards, MQTT, APIs, and other integrations.

Figure 10.12 shows some Node-RED node groups and examples of nodes within them.

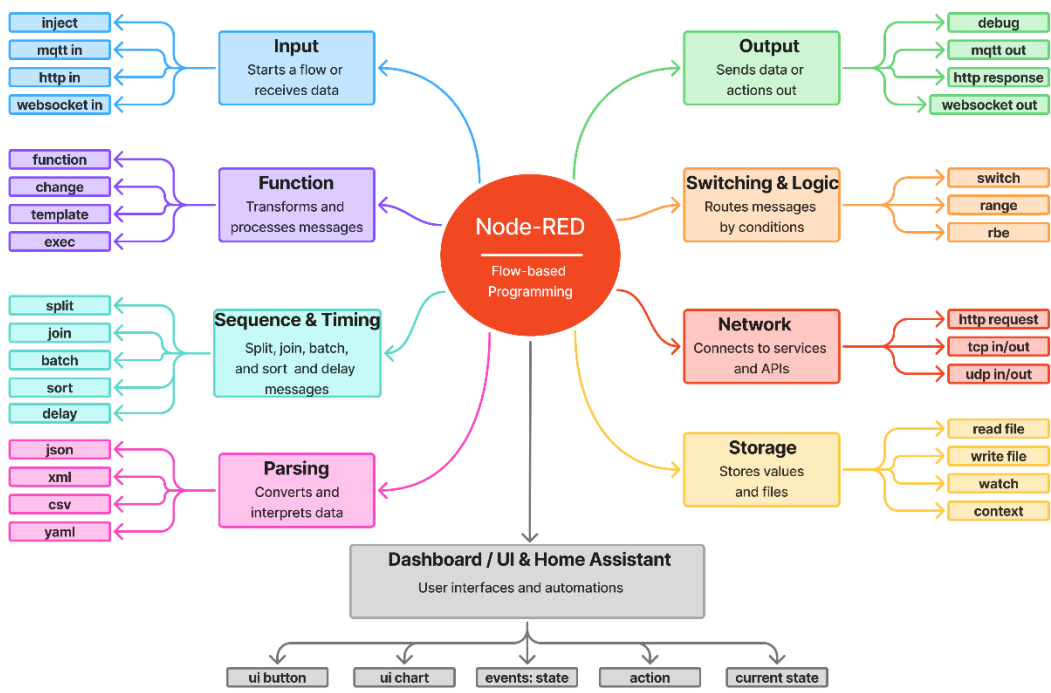


Figure 10.12 – Node-RED node groups

The diagram in Figure 10.12 shows that Node-RED provides a broad palette of nodes that can be combined to create automation flows using different resources and functionalities. The Node-RED group library is available here: <https://flows.nodered.org/>.

Table 10.2 shows some nodes examples and how they can be used to implement automation.

Node	Purpose	Example Use
events: state	Listens for entity changes	Motion sensor turns on
current state	Reads current entity state	Check home mode
action	Calls Home Assistant service	Turn on a light
function	Runs JavaScript logic	Build a custom payload
debug	Displays message data	Troubleshoot flow output

Table 10.2 – Node-RED node examples

In the next section, we will compare native automations, AppDaemon, and Node-RED to choose the right tool for each automation challenge.

Orchestrating external automation logic: choosing the right tool

At this point in the chapter, we have explored three different ways to build advanced automation logic in Home Assistant: native Home Assistant automations, AppDaemon, and Node-RED. Each one can solve automation problems, but they are not designed for the same type of task. Choosing the right tool is important because it affects how easy the automation is to build, debug, maintain, and expand over time.

This section helps you decide when to use each tool and how they can work together in a layered automation architecture.

Comparing the three automation approaches

Each tool has different features. Native automations are close to Home Assistant and are ideal when the automation logic is directly related to entity states, triggers, conditions, and services. AppDaemon gives you the flexibility of Python, making it useful when you need more control over the behavior of the automation. Node-RED provides a visual workflow environment, which is helpful when you debug your automation.

Table 10.3 summarizes the main features of each automation tool.

Tool	Best Use Case	Main Strength	Main Limitation
Native Automations	Simple, fast, UI-driven logic	Easy to create and manage inside Home Assistant	Can become difficult to maintain when logic grows
AppDaemon	Complex logic and reusable automation applications	Uses Python and structured programming	Requires Python coding knowledge
Node-RED	Visual flows and integrations across systems	Excellent visibility and debugging	Large flows can become hard to organize

Table 10.3 – Automation tool comparison

A good advanced automation system does not need to use only one of these tools. From my experience, the best approach is to combine them considering their best use case and strengths according to *Table 10.3*.

Automation tool decision guidelines

The following decision process can help you select the right tool:

1. Start with native Home Assistant automations. If the automation is simple, easy to describe, and does not require complex reusable logic, keep it native.
2. Move to AppDaemon when the automation requires structured programming. If you are repeating the same YAML logic many times, need advanced timers, or want to create reusable automation modules, AppDaemon is a better fit.
3. Use Node-RED when the workflow is easier to understand visually or when it connects multiple external systems. If you need to inspect messages, transform payloads, or build a multi-step integration flow, Node-RED is often the most practical option.

The following diagram in *Figure 10.13* summarizes the decision process:

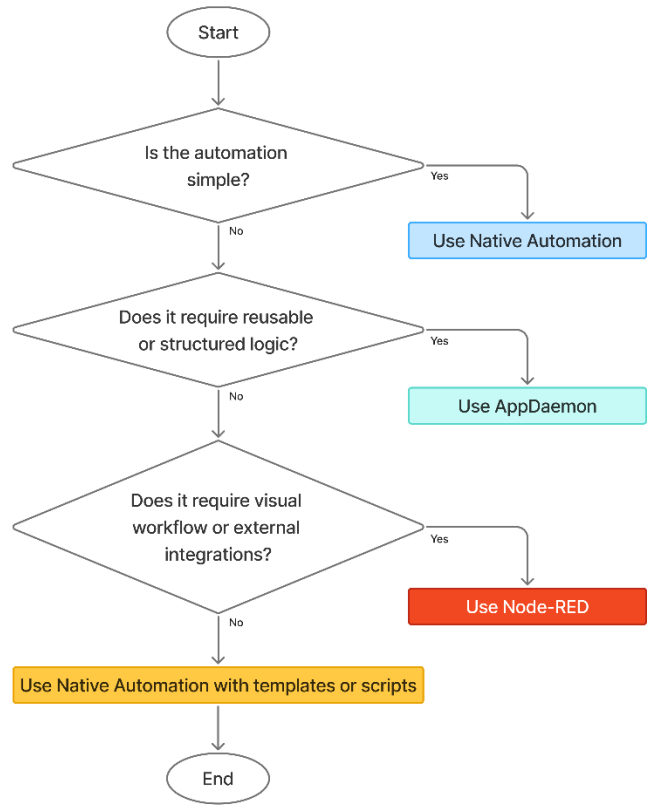


Figure 10.13 – Choosing the right automation tool

If you answer no to all the three questions in the *Figure 10.13* flowchart then it will be best to use Home Assistant Automation with templates and scripts.

Hybrid architecture: combining the tools

In advanced Home Assistant systems, the best solution is often not choosing only one tool, but combining them. Each tool can be responsible for a different layer of the automation system.

A recommended architecture is presented in *Table 10.4*:

Layer	Tool	Responsibility
Orchestration layer	Home Assistant	Entity states, services, dashboards, state machines, scenes, and basic automations

Layer	Tool	Responsibility
Workflow layer	Node-RED	Visual flows, integrations, routing, and data transformations
Logic layer	AppDaemon	Reusable Python logic, and complex automation applications

Table 10.4 – Automation tools architecture

In this model, Home Assistant remains the central platform. It stores states, exposes services, manages integrations, and provides the user interface. Node-RED acts as a workflow engine when data must move through several steps or systems. AppDaemon acts as a logic engine when the automation needs structured programming and reusable behavior.

In the next subsection, we will explore an example with this hybrid architecture.

Practical example: hybrid security automation using Home Assistant, Node-RED, and AppDaemon

In this example, we will build a hybrid security automation that combines Home Assistant, Node-RED, and AppDaemon. The goal is to detect when a door is opened, wait for possible motion inside the house, evaluate the current home status, and then decide whether the system should ignore the event, send a normal notification, or trigger a higher-priority security alert.

Instead of creating all the logic in a single Home Assistant automation, the system is divided into three layers as shown in Table 10.5:

Layer	Tool	Responsibility
State Layer	Home Assistant	Stores helpers, groups, and the home status state machine
Flow Layer	Node-RED	Detects the door and motion sequence and creates a security decision event
Logic/action layer	AppDaemon	Processes the decision, updates status, sends notifications, and turns on lights

Table 10.5 – Three layers automation tool

In the next sections, you will understand how each layer was implemented.

Note

All code implemented for this example is available in the book's GitHub repository. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface*.

State layer - Home Assistant helpers and groups

The following helpers and groups presented in *Table 10.6* were created to support the example.

Entity	Type	Purpose
input_select.home_status	Dropdown helper	Stores the effective home mode used by Node-RED
input_select.home_status_override	Dropdown helper	Allows the user to force special modes such as party or exempt
input_select.security_status	Dropdown helper	Stores the current security state updated by AppDaemon
input_text.last_security_message	Text helper	Stores the last security message for dashboard visibility
group.all_doors	group	Represents all monitored door sensors
group.all_motion_sensors	group	Represents all monitored motion sensors

Table 10.6 – Helpers and groups created in Home Assistant

The `input_select.home_status` helper is automatically updated by a Home Assistant state machine. It has 5 states in total: home, away, night, party and exempt. Home, away and night are established automatically by sensors and other conditions like time of the day while party and exempt are configured by the user using the helper explained next.

The `input_select.home_status_override` helper gives the user manual control over the automatic home status state machine. This is useful when normal automation logic should be temporarily changed. For example, during a party, the doors and motion sensors may activate frequently, but these events should not trigger high-priority alerts. An automation in Home

Assistant can be implemented to automatically reset the party and exempt states back to auto mode after 6 hours.

The `input_select.security_status` helper is updated by AppDaemon after the security decision is processed. This makes the final state visible in Home Assistant dashboards.

Flow layer – Node-RED description

Figure 10.14 shows the flow implemented in Node-RED in this example.

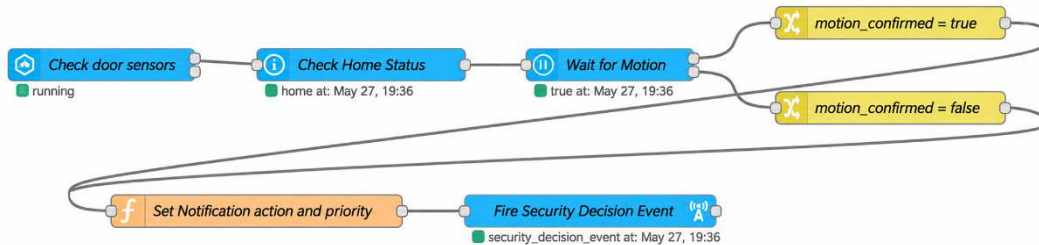


Figure 10.14 – Node-RED flow

The flow starts with the `Check door sensors` node. This node monitors the grouped door sensor entity. When one of the doors opens, the flow continues to the `Check Home Status` node, which reads the current value of `input_select.home_status`.

After reading the home status, the flow moves to the `Wait for Motion` node. This node waits for `groups.all_motion_sensors` to become active. The node has two outputs:

- Output 1: Motion was detected before the timeout
- Output 2: Timeout occurred and no motion was detected

The next node, `Set Notification action and priority`, evaluates the home status and motion flag. It creates a structured object called `msg.security_decision`. This object is then sent to Home Assistant using the `Fire Security Decision Event` node. The event name used by this flow is called `security_decision_event`. This event is the handoff point between Node-RED and AppDaemon.

The function node `Set Notification action and priority` implements the Security Decision Matrix summarized in Table 10.7.

Home Status	Motion Confirmed	Action	Priority	Security State	Message
exempt	Yes or No	ignore	none	exempt	Door opened, but security logic is exempt
party	Yes or No	notify	low	watching	Door opened during party mode
away	Yes	alert	critical	alarm	Door opened while away, followed by indoor motion
away	No	alert	high	warning	Door opened while away, but no indoor motion was detected
night	Yes	alert	high	warning	Door opened at night, followed by indoor motion
night	No	notify	normal	watching	Door opened at night, but no indoor motion was detected
home	Yes	notify	normal	watching	Door opened, followed by indoor motion
home	No	notify	normal	watching	Door opened, but no indoor motion was detected

Table 10.7 – Security decision matrix

As previously commented, the Fire Security Decision Event node sends a structured event payload to AppDaemon. *Table 10.8* shows the fields used in the payload message and a possible example for each field.

Field	Purpose	Example
home_status	Defines the current home mode	away
motion_confirmed	Indicates if motion occurred after the door opened	true
action	Defines the type of action AppDaemon should perform	alert
priority	Defines the urgency of the event	critical
security_state	Defines the value written to input_select.security_status	alarm
reason	Explanation for the message	Door opened while away, followed by indoor motion
detected_at	Records when the decision was created	ISO timestamp

Table 10.8 – Security decision event payload message

Logic layer – AppDaemon processing

AppDaemon listens to the `security_decision_event` generated by Node-RED. When the event is received, the Python app extracts the values from the event payload and performs the final actions.

Two files were created: `apps.yaml` and `security_state_machine.py`. The main Python functions created are the following:

- `security_decision_received()`: Main event handler
- `set_security_status()`: Updates `input_select.security_status`
- `set_last_security_message()`: Updates `input_text.last_security_message`
- `send_notification()`: Sends normal or low-priority phone notifications
- `send_high_priority_alert()`: Sends urgent phone notifications
- `turn_on_security_lights()`: Turns on selected lights and switches

The notification recipients and security lights are configured in `apps.yaml`, not hardcoded in the Python file. This makes the application reusable. If a phone or light changes, only `apps.yaml` needs to be updated.

Testing the automation

There are several test cases that can be used to test the hybrid automation. *Table 10.9* can be used to test it.

Testcase	Home Status	Door Opened	Motion Detected	Expected Security State	Expected Action
1	away	Yes	Yes	alarm	Critical alert and security lights on
2	away	Yes	No	warning	High-priority alert
3	night	Yes	Yes	warning	High-priority alert
4	night	Yes	No	watching	Normal notification
5	party	Yes	Yes or No	watching	Low-priority notification
6	exempt	Yes	Yes or No	exempt	Ignore event
7	home	Yes	Yes	watching	Normal notification

Table 10.9 – Example test cases

The test should be done in layers. First, confirm that the Home Assistant helpers `input_select.home_status` and `input_select.home_status_override` update correctly. Next, use Debug nodes in Node-RED to inspect `msg.home_status`, `msg.motion_confirmed`, and `msg.security_decision`. Finally, confirm that `AppDaemon` receives the event and updates `input_select.security_status` and `input_text.last_security_message`.

This example demonstrates how a hybrid automation can be designed using clear responsibilities. This completes the section about choosing the right automation tool. With the hybrid workflow now complete, the next section will focus on making this automation more resilient.

Designing resilient home automation systems

Home automation systems work well when everything is connected, responsive, and predictable. However, smart homes are not perfect environments. Devices may disconnect, Wi-Fi or Zigbee networks may become unstable, sensors may report incorrect values, and cloud services may become temporarily unavailable. You may believe your system is automated, but it silently fails without warning.

For this reason, advanced automation design should also consider what the system should do when something fails. A resilient automation system can detect problems, retry actions, use fallback paths, and notify the user when manual intervention is required.

Resilience is especially important for automations related to safety, security, lighting, HVAC, garage doors, water leak detection, and alerts. These automations should be designed with extra care because failure can create inconvenience, energy waste, or even safety risks.

Resilient automations are designed to keep working even when devices, sensors, or networks behave unexpectedly. Instead of assuming that every command succeeds, the automation verifies results, retries when needed, uses fallback actions, and notifies the user when attention is required. *Table 10.10* summarizes the main design principles you can use to build reliable Home Assistant automations.

Design Principle	Description	Example	Why It Improves Resilience
Retry Mechanism	Try the action again if the first attempt may have failed.	Repeat <code>switch.turn_on</code> three times with a short delay.	Handles temporary communication issues with Wi-Fi, Zigbee, MQTT, or slow devices.
Timeout Handling	Stop waiting after a defined period.	Wait 60 seconds for a garage door sensor to report closed.	Prevents automations from getting stuck when a device or sensor does not respond.

Design Principle	Description	Example	Why It Improves Resilience
Fallback Logic	Use an alternate device or path when the main one fails.	If a Zigbee light fails, turn on a Wi-Fi lamp.	Keeps the automation useful even when the preferred device is unavailable.
Feedback-Based Confirmation (Closed-loop)	Confirm the result using another sensor.	Use an illuminance sensor to confirm that a light actually turned on.	Detects cases where the command was sent but the physical action did not happen.
Monitoring and Alerts	Notify the user when something important fails.	Send a mobile notification if a critical device is unavailable.	Makes failures visible instead of allowing the system to fail silently.

Table 10.10 – Reliable home automation design principles

The flowchart presented in *Figure 10.15* represents a generic automation that includes all the design principles presented in *Table 10.10*.

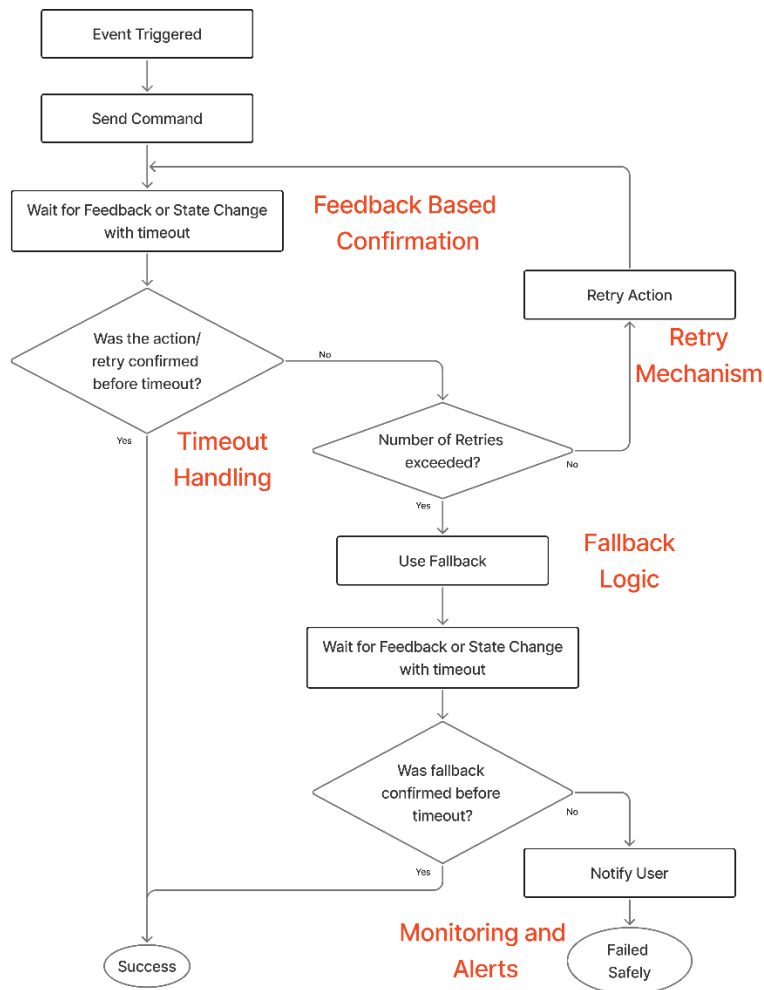


Figure 10.15 – Resilient automation flow with key design principles

In resilient design, the goal is to handle failures safely. If the original action and fallback action both fail, the automation should notify the user and finish in a controlled failure state. This final condition is considered failed safely, because the system recognized the problem, avoided getting stuck, and made the failure visible.

These principles help transform a basic automation into a more reliable control flow. In simple cases, one or two techniques may be enough. For critical automations, such as garage doors, security alerts, leak detection, or nighttime lighting, combining confirmation, timeout handling, retries, fallback logic, and notifications provides a much safer and more predictable result.

Summary

In this chapter, you learned how to move beyond basic Home Assistant automations by using structured automation patterns. State machines, sequences, and conditional flows were introduced as ways to organize complex logic, separate sensing from decision-making, and make automations easier to maintain as your system grows.

You also explored AppDaemon as a Python-based automation engine for Home Assistant. The chapter showed how AppDaemon can create reusable automation applications, manage timers, pass configuration through `apps.yaml`, and build simple dashboards with HADashboard.

Next, you learned how Node-RED can be used as a visual workflow tool for Home Assistant. Node-RED flows help with debugging, multi-step automations, integrations, and data transformations. After comparing native automations, AppDaemon, and Node-RED, the chapter demonstrated a hybrid security automation where each tool had a clear responsibility.

Finally, you explored resilient automation design, including retries, timeout handling, fallback logic, feedback-based confirmation, monitoring, and alerts. In *Chapter 11, Extending Home Assistant with HACS*, we will explore community-developed integrations and learn how HACS can extend Home Assistant beyond the official integration catalog.

11

Extending Home Assistant with HACS

As your Home Assistant system grows, you will eventually reach a point where the official integrations, default dashboard cards, and built-in tools do not cover every device, workflow, or visualization you want to create. This is where HACS, the Home Assistant Community Store, becomes important. HACS gives users a structured way to discover, install, and maintain community-developed integrations, frontend cards, themes, and other extensions. Instead of manually copying files from different GitHub repositories, HACS centralizes much of this process and makes it easier to expand Home Assistant beyond its default capabilities.

This chapter builds naturally on the previous chapters. Earlier, we explored Home Assistant architecture, custom integrations, Zigbee, MQTT, ESPHome, AppDaemon, Node-RED, and resilient automation design. Those topics showed how powerful Home Assistant becomes when it is extended with external logic, custom devices, and advanced workflows. HACS is the next step because it connects your system to the broader Home Assistant community, where many advanced features, dashboard components, and experimental integrations are first introduced.

In this chapter, we will explore how to use HACS safely and effectively, how to evaluate community integrations before installing them, how custom integrations are structured, and how to maintain and update your HACS ecosystem.

By the end of this chapter, you will be able to navigate the HACS ecosystem, install and configure advanced community integrations, evaluate their reliability, troubleshoot common problems, understand the basic structure of custom components, and maintain your HACS environment safely.

The main topics covered are:

- Introduction to HACS: the Home Assistant Community Store
- Installation, Configuration, and Troubleshooting HACS Integrations
- Curated Advanced HACS Contents: Beyond the Official
- Understanding custom component structure and development basics
- Maintaining and updating your HACS ecosystem

Technical requirements

To follow this chapter, you should have:

- A working Home Assistant installation with administrator access.
- Basic familiarity with Home Assistant integrations, entities, services, and dashboards.
- Experience navigating Settings, Devices & services, Add-ons, and the Home Assistant file structure.
- A GitHub account, since HACS and most community integrations are distributed through GitHub repositories.
- Basic knowledge of YAML, especially if a HACS integration or frontend card requires manual configuration.
- A recent backup of your Home Assistant system before installing or updating community integrations.

Introduction to HACS: the Home Assistant Community Store

HACS stands for Home Assistant Community Store. It is a community-driven system that helps users discover, install, update, and manage custom content created by the Home Assistant community.

It is important to explain early that HACS is not a replacement for official Home Assistant integrations. Official integrations are included with Home Assistant Core and follow Home Assistant's formal development, review, and release process. HACS content, on the other hand, is created and maintained by individual developers or community groups. Some projects are mature, popular, and actively maintained, while others may be experimental, niche, or no longer updated.

This distinction is important because HACS gives you more flexibility, but it also gives you more responsibility. When installing community content, you must evaluate whether the repository is active, documented, compatible with your Home Assistant version, and trusted by the community.

HACS manages several types of community content. *Table 11.1* lists the categories managed by it.

HACS category	Purpose	Example use case
Integrations	Add new backend functionality to Home Assistant	Connect to a device, service, or API not officially supported
Dashboard cards	Add custom dashboard components	Build advanced dashboards with custom controls
Themes	Change the visual appearance of Home Assistant	Create a dark, tablet-friendly, or brand-specific interface
Templates	Share reusable template logic	Add complex sensors, notifications, or UI helpers
AppDaemon apps	Share Python-based automation apps for AppDaemon	Install reusable advanced automation logic that runs through AppDaemon

Table 11.1 – Categories managed by HACS

Now that you understand what HACS is and what it manages, let's move into the practical workflow. In the next section, we will install, configure, and troubleshoot HACS content safely before exploring specific advanced examples.

Installation, configuration, and troubleshooting HACS contents

Before installing community integrations, HACS itself must be added to Home Assistant. For Home Assistant OS installation, the simplest method is to add the official HACS repository to the Apps Store, install the HACS app, and then complete the integration setup from Home Assistant. These are the installation steps:

1. Go to **Settings | Apps | Install app**.
2. Open the menu in the upper-right corner and select **Repositories**.
3. Add the official HACS repository URL from the HACS documentation. Currently it is: <https://github.com/hacs/addons>
4. Scroll down to find the option **Get HACS** as presented on *Figure 11.1*. Click on it.



Figure 11.1 – HACS installation option using HAOS

5. In the next screen click **Install**.
6. Start the app
7. Restart Home Assistant. Perform a full Home Assistant restart, not only a quick reload.
8. Go to **Settings | Devices & services | Add integration**.
9. Search for HACS and complete the GitHub authorization according to instructions here: <https://hacs.xyz/docs/use/configuration/basic/#to-set-up-the-hacs-integration>.

After this process, HACS will appear in the Home Assistant sidebar and can be used to install community integrations, frontend cards, and themes. Clicking HACS in the sidebar gives you access to all the content managed by HACS as presented in *Table 11.1*. *Figure 11.2* shows a snapshot of the HACS main screen.

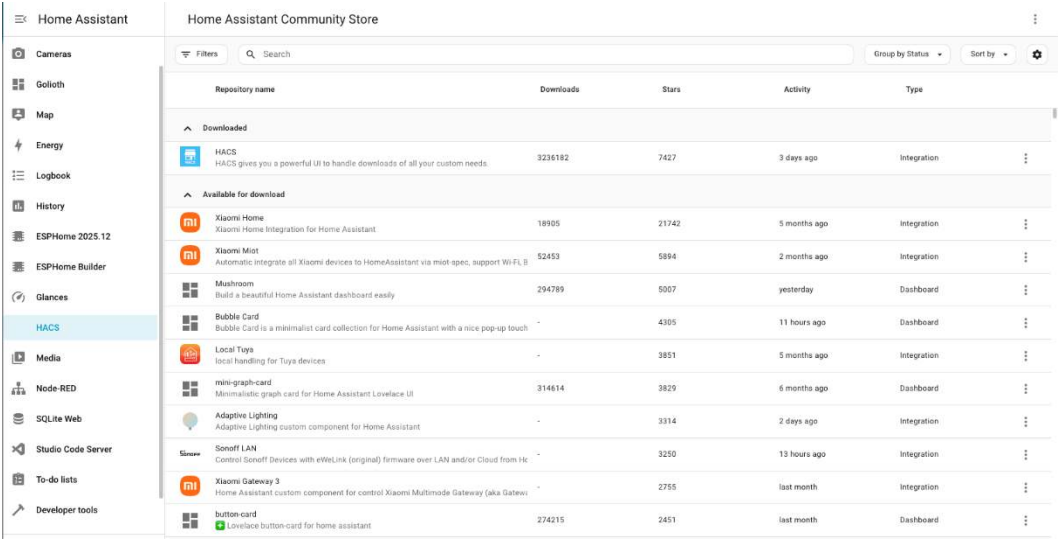


Figure 11.2 – HACS main screen

You can filter the HACS content by type with the options: Appdaemon, dashboard, Integration, Template and Theme. There is a search field, and you can group the content by status or content type. You can also sort the content by different options such as downloads, stars, or activity, among others.

Installing HACS content

Installing HACS content is usually straightforward, but it is important to understand that HACS does not configure it for you automatically. HACS mainly handles discovery, downloading, and updates. After the files are installed, Home Assistant still needs to load the integration, validate its dependencies, and create the entities or services that the content provides.

The typical installation process starts inside HACS, where you search for the content you want to install. Before downloading anything, always review the documentation, GitHub repository, recent issues, and compatibility notes. A good approach is to review the number of downloads and stars that the content has. A high number of downloads and stars can indicate adoption, but it does not guarantee reliability.

After downloading the HACS content, restart Home Assistant as best practice. This allows the custom component to be loaded from the custom_components directory. Once Home Assistant restarts, the integrations can be added from Settings | Devices & services, while others require YAML configuration in configuration.yaml or another included YAML file. *Table 11.2* summarizes the HACS content type and what happens after the download.

HACS content type	What happens after download
Integration	Restart Home Assistant, then add or configure it from Settings → Devices & services
Dashboard card	Refresh the browser, then use the custom card in a dashboard
Theme	Make sure themes are enabled in configuration.yaml, then select the theme from the user profile
Template or script	Follow the repository instructions; may require YAML changes

Table 11.2 – What happens after downloading each HACS content type

Configuring HACS content and confirming that it works

After Home Assistant restarts, go to Settings | Devices & services and click Add integration if you want to install an integration. Search for the installed integration. If the integration supports a configuration flow, Home Assistant will display a setup form where you can enter required information such as device IP address, username, password, API token, or other options.

Some HACS integrations still require YAML configuration. In that case, the integration may not appear in the UI until the YAML configuration is added and Home Assistant is restarted again. This is why reading the documentation is important.

The next step after installation and configuration is to verify that the content is working. Depending on what you installed, there are different steps to follow. *Figure 11.3* shows screenshot examples of how this can be verified using Home Assistant. For backend integrations, go to Settings | Devices & services and open the integration page (*Figure 11.3 - 1*). Confirm that the device appears, and that entities are available. For frontend cards, open the dashboard editor and confirm that the custom card can be selected or used in YAML mode (*Figure 11.3 - 2*). For themes, go to the user profile or theme selection area and confirm that the new theme is available (*Figure 11.3 - 3*).

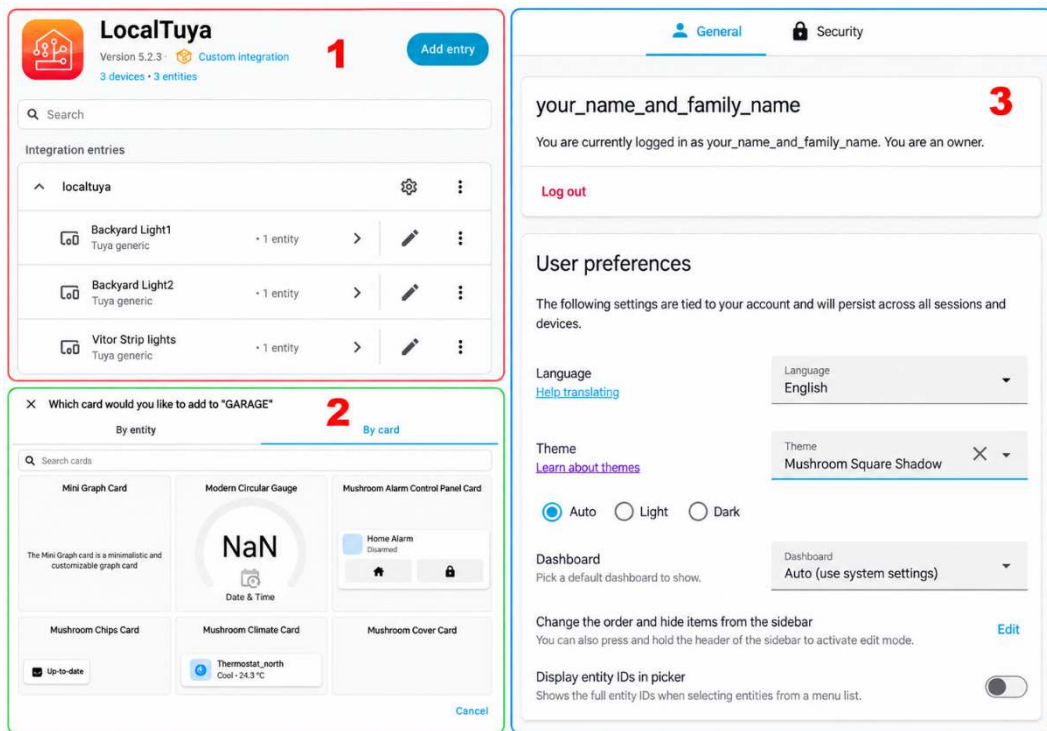


Figure 11.3 – Screenshot examples of HACS installation confirmation

Troubleshooting HACS content

When HACS content does not work as expected, start by identifying what type of content was installed. Integrations, frontend cards, themes, and templates fail in different ways. Integrations usually require a Home Assistant restart and should be checked in Settings | Devices & services. Frontend cards usually require a browser refresh, cache clearing, or dashboard resource verification.

Themes may require the Home Assistant frontend theme configuration to be enabled in YAML. If the theme does not appear, confirm that frontend themes are enabled and restart Home Assistant.

A simple troubleshooting sequence is:

1. **Confirm the content type:** Check whether you installed an integration, dashboard card, theme, or template.
2. **Restart or refresh as required:** Restart Home Assistant for integration content. Refresh the browser or clear cache for cards and themes.

3. **Check the Home Assistant logs:** Go to **Settings | System | Logs** and search for the integration name, repository name, or custom component folder name.
4. **Verify configuration:** For integrations, check **Settings | Devices & services** and confirm that entities were created. For cards, check dashboard resources and card YAML. For themes, confirm that themes are enabled.
5. **Review the repository documentation:** Check the README, release notes, and open GitHub issues for known problems or breaking changes.
6. **Roll back or remove if needed:** If an update breaks the system, restore a backup, roll back the HACS content if possible, or remove the custom content manually.

These steps are summarized in the workflow presented on *Figure 11.4*.

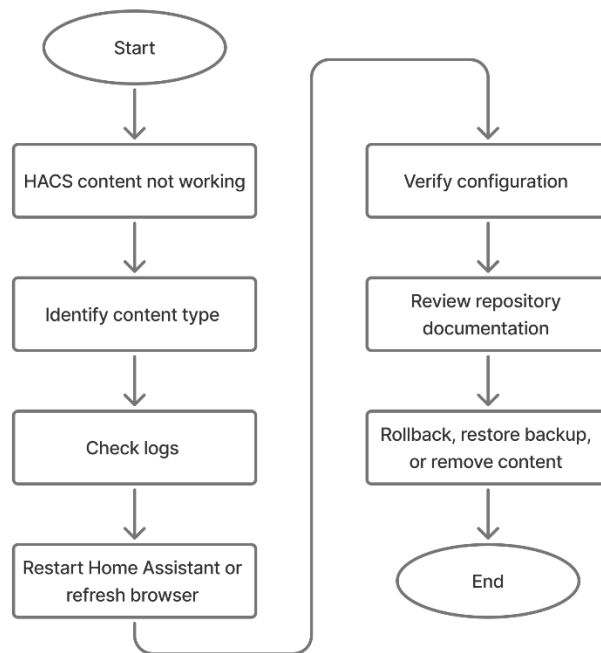


Figure 11.4 – HACS troubleshooting flow

By following the process described in *Figure 11.4*, you can take advantage of community content and reduce the risk of disrupting your smart home system.

Curated advanced HACS content: beyond the official

HACS contains many community projects, and those projects change over time. For this reason, instead of listing the most used or most downloaded content, a better approach is to teach you how to think in categories: what problem the integration solves, how it fits into your Home Assistant architecture, and how much risk it adds to your system.

Recommended categories for advanced users

Before choosing what to install from HACS, it is useful to think about the role each community project will play in your Home Assistant system. Some HACS projects improve dashboards, others add monitoring capabilities, support unsupported devices, simplify automation logic, or change the visual appearance of the interface. Organizing HACS content into categories helps you make better decisions, avoid unnecessary installations, and connect each integration to a clear purpose. *Table 11.3* presents practical categories that are especially useful for advanced users and relates them to the projects developed throughout this book.

Category	Why it matters	Example used in this book	What to evaluate before installing
Dashboard cards	Improve visualization and control	Advanced dashboards for ESPHome devices, cameras, Zigbee, MQTT, and AI suggestions	Browser compatibility, frontend resource loading, recent updates
Monitoring integrations	Improve reliability	Track unavailable entities, low batteries, failed integrations, or noisy sensors	Maintenance status, entity count impact, alert configuration
Data visualization	Improve analysis	Display energy, temperature, MQTT history, or device behavior	Recorder impact, long-term statistics compatibility
Device-specific integrations	Support unsupported devices	Add niche Wi-Fi, BLE, Zigbee, cloud, or local API devices	Repository activity, issues, API dependency, HA compatibility

Category	Why it matters	Example used in this book	What to evaluate before installing
Automation helpers	Improve complex logic	Reusable patterns for notifications, schedules, or logic helpers	Whether native HA helpers can already solve the problem
Themes	Improve usability	Tablet dashboards, mobile-friendly views, or wall-panel layouts	Theme maintenance, readability, dark/light mode behavior

Table 11.3 – Practical HACS content categories

Before exploring specific examples, it is important to remember that HACS content can serve very different purposes. The examples below are not intended to be a fixed list of required installations. Instead, they show how to evaluate common HACS categories and how each type of project can support the advanced Home Assistant systems developed throughout this book.

Example 1 – Advanced dashboard cards

Frontend cards are one of the most popular uses of HACS because they change how Home Assistant information is presented to the user. Official Home Assistant cards are stable and well supported, but custom cards can create richer dashboards, compact controls, and specialized views that are better suited for advanced systems developed by you.

For the projects developed in this book, HACS dashboard cards can be used to create a central control panel that combines data from Zigbee, MQTT, ESPHome, cameras, and others.

Figure 11.5 shows a collection of dashboard cards created using different HACS projects.

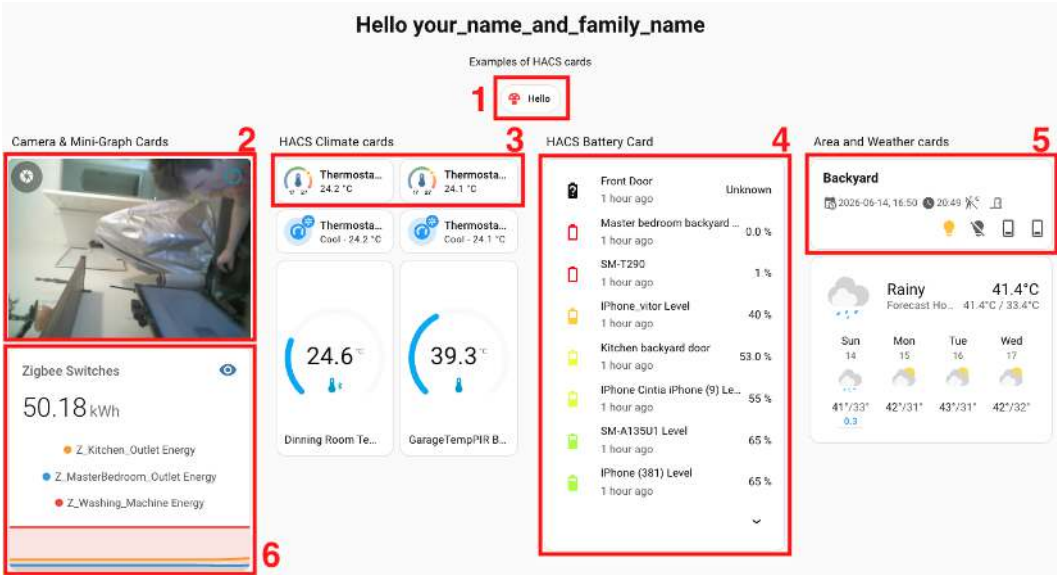


Figure 11.5 – HACS dashboard cards sample

Among the examples of dashboard cards installed and presented in *Figure 11.5* are Mushroom (1), Advanced Camera Card (2), Ring Tile Card (3), Battery State Card (4), Minimalist Area cards (5), mini-graph card (6).

Example 2 – System monitoring integrations

As your Home Assistant system grows, reliability becomes more important than the number of devices you add. A mature system may include hundreds or thousands of entities, and some of them will eventually become unavailable, stop updating, report low battery, or generate excessive database entries.

Monitoring-focused HACS integrations can help identify these issues earlier. They can support the same reliability strategy introduced in *Chapter 10*, where resilient automation design used feedback, retries, timeout handling, and monitoring.

A monitoring integration can help you identify common issues presented in *Table 11.4*.

Monitoring target	Application	Application Action
Unavailable entities	Detect broken devices or integrations	Notify the user if a critical sensor is unavailable

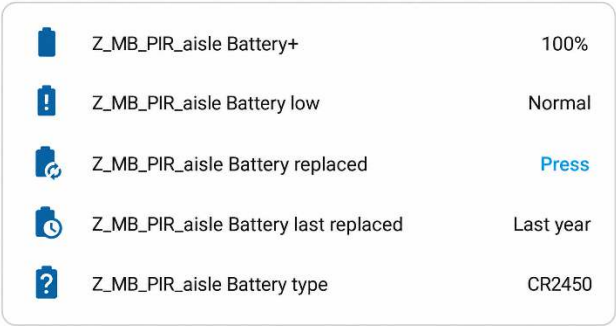
Monitoring target	Application	Application Action
Low battery devices	Prevent silent automation failures	Create weekly battery maintenance reports
Failed integrations	Identify unstable services or cloud dependencies	Restart, disable, or investigate the integration
High-update entities	Reduce recorder/database load	Exclude noisy entities or create filtered sensors
Stale sensors	Detect sensors that stopped reporting	Alert if a sensor has not changed in 24 hours
Automations not running	Validate expected behavior	Create a watchdog notification

Table 11.4 – Monitoring integration applications

Some examples of integrations that can be used for system monitoring are presented as follows:

- Watchman (<https://github.com/dummylabs/thewatchman>) : provides a report of missing entities, unavailable entities, and unavailable services/actions referenced in YAML. It helps detect broken automations caused by renamed, deleted, or unavailable entities.
- Spook (<https://spook.booo/>) : provides Home Assistant repairs and helps find orphaned entities and hidden problems. It also supports advanced system checks. It is a system health toolbox for finding configuration and entity issues.
- Battery Notes (<https://github.com/andrew-codechimp/HA-Battery-Notes/>): monitors battery type, replacement dates, and battery maintenance notes. Helps track maintenance for battery-powered Zigbee, BLE, and wireless sensors

After installing Battery Notes integration, for example, you can add new entities related to battery in the dashboard. *Figure 11.6* shows one of the Zigbee Motion Sensors battery entities information created by Battery Notes integration.



	Z_MB_PIR_aisle Battery+	100%
	Z_MB_PIR_aisle Battery low	Normal
	Z_MB_PIR_aisle Battery replaced	Press
	Z_MB_PIR_aisle Battery last replaced	Last year
	Z_MB_PIR_aisle Battery type	CR2450

Figure 11.6 – Battery Notes integration entities

System monitoring integrations help you keep the current system reliable, but advanced systems often need to support devices that Home Assistant does not officially recognize. That is where custom device integrations become useful, allowing you to extend the system while still applying the same evaluation and validation process. This is what we will see in the next subsection.

Example 3 – Custom device integrations

Not every smart home device has an official Home Assistant integration. Sometimes, a community developer creates support for a device before official support exists. In other cases, the device is too niche, experimental, or region-specific to justify a core integration.

You may find custom integrations for new smart plugs, BLE sensors, local APIs, unofficial cloud APIs, experimental ESP32 projects, or devices that expose useful data but are not officially supported yet.

However, this is also where you need to be most careful. A custom integration runs inside Home Assistant and can affect stability, performance, and security. Before installing one, check the repository activity, open issues, documentation quality, release history, and compatibility with the current Home Assistant version. HACS documentation also notes that custom repositories need to follow a known structure, and HACS publishing has requirements for supported repository types such as integrations, plugins, AppDaemon apps, templates, and themes.

Some examples of custom device integrations are listed as follows:

- **Alexa Media Player** (https://github.com/alandtse/alexa_media_player/): Adds Echo/Alexa devices as media players and enables voice announcements for alerts, reminders, and automation status messages. It may remain custom because it relies on unofficial Amazon behavior that can change without notice.

- LocalTuya (<https://github.com/rospogrigio/localtuya>): Provides local control for many Tuya-based Wi-Fi devices, such as plugs, switches, and lights. It is useful when local control is preferred over cloud-only control, but it may remain custom because Tuya's ecosystem and cloud services are central to its platform.
- SonoffLAN (<https://github.com/AlexxIT/SonoffLAN>): Adds local or hybrid control for Sonoff/eWeLink relays, plugs, switches, and similar devices. It can replace or complement eWeLink-created entities by exposing devices through this custom integration.
- Xiaomi Miot (<https://github.com/al-one/hass-xiaomi-miot>): Expands support for Xiaomi, Mijia, and Miot-based devices such as air purifiers, fans, humidifiers, sensors, and lights. It may remain custom because Xiaomi's ecosystem includes many region-specific devices and cloud behaviors.

Figure 11.7 shows LocalTuya and SonoffLAN integration devices I have installed in my home. The main advantage I found to have them installed instead of the vendor application is the ability to use the devices without having them connected to the cloud.

	Device	Area	Integration	Manufacturer	↑ Model
	Front Light	Front House	Sonoff	SONOFF	BASICR2
	MB Lamp	Master Bedroom	Sonoff	SONOFF	MINI
	MB Fan	Master Bedroom	Sonoff	SONOFF	MINI
	Vitor plug	kid bedroom 1	Sonoff	SONOFF	S40TPB
	TV	Family Room	Sonoff	SONOFF	S40TPB
	Mb chairlamp	Master Bedroom	Sonoff	SONOFF	S40TPB
	Coffeemaker		Sonoff	SONOFF	S40TPB
	Vitor Strip lights		LocalTuya	Tuya	Tuya generic (eb5e)
	Backyard Light1		LocalTuya	Tuya	Tuya generic (eb8b)
	Backyard Light2		LocalTuya	Tuya	Tuya generic (ebd4)

Figure 11.7 – Sonoff and LocalTuya devices installed in my home

Example 4 – Themes for usability and wall-panel dashboards

Themes are another practical category of HACS content because they improve the usability of Home Assistant without changing device logic. Themes define the visual language of the dashboard. This becomes especially important for wall-mounted tablets, mobile dashboards, and shared family interfaces where readability, contrast, and consistency matter as much as technical capability.

Here are some useful theme examples for your reference:

- Catppuccin (<https://github.com/catppuccin>): Provides a soft, polished color palette with light and dark variants, making it useful for clean mobile dashboards, room dashboards, and general-purpose control panels.
- Noctis (<https://github.com/aFFekopp/noctis>): Creates a dark blue interface that works well in low-light environments, making it useful for night dashboards, bedroom wall panels, and security dashboards.
- iOS Themes (<https://github.com/basnijholt/lovelace-ios-themes/>): Gives Home Assistant a HomeKit-style appearance with light and dark options, making it useful for dashboards designed for users familiar with Apple-style interfaces.
- UI Lovelace Minimalist (<https://github.com/UI-Lovelace-Minimalist/UI>): Provides a complete minimal dashboard style with a modern app-like feel, making it useful for advanced tablet dashboards and polished smart home control centers.
- Mushroom Themes (<https://github.com/piitaya/lovelace-mushroom-themes>): Adjusts the visual style of dashboards built with Mushroom cards, making it useful for simple, modern dashboards using Mushroom card layouts.

The examples in this section show how HACS can extend Home Assistant through dashboards, monitoring tools, device support, and themes. After learning how to evaluate these community projects from a user perspective, the next step is to understand what exists behind them. In the next section, we will look at the basic structure of a custom component, how its files are organized, and how Home Assistant loads it, giving you a clearer foundation for troubleshooting HACS integrations or eventually creating your own.

Understanding custom component structure and development basics

In *Chapter 1*, we introduced the idea of extending Home Assistant by creating a custom integration for a dual measurement sensor. That example showed how a device that publishes

temperature and presence information through MQTT can be represented in Home Assistant as a more organized integration instead of a collection of separate YAML-defined entities. In this chapter, we return to that idea from a different perspective: how a personal custom integration can be structured, documented, and prepared for distribution through HACS.

A custom integration can be installed manually by copying its folder into the `custom_components` directory and restarting Home Assistant. This is useful for personal testing, but it is not ideal for sharing with others. If you want other users to install, update, and manage the integration more easily, the project should be organized as a GitHub repository that HACS can recognize.

A simplified HACS-compatible repository for the dual measurement sensor integration could look like this:

```
your_integration_repository/
├── custom_components/
│   └── your_integration/
│       ├── __init__.py
│       ├── manifest.json
│       ├── config_flow.py
│       ├── const.py
│       ├── sensor.py
│       ├── binary_sensor.py
│       └── translations/
│           └── en.json
├── README.md
├── hacs.json
└── LICENSE
```

Compared with a local custom integration, a HACS-ready integration has the same codebase and the files still live under `custom_components/integration_name/`. What changes is the project packaging. A HACS repository should place the integration inside a GitHub repository using the expected `custom_components` structure and include repository-level files such as `README.md`, `hacs.json`, and a `LICENSE`. The `README.md` explains how the integration works, while `hacs.json` provides HACS-specific metadata.

An example of the `hacs.json` file content for the Dual Measurement Sensor is the following:

```
{
  "name": "Dual Measurement Sensor",
  "content_in_root": false,
```



```
"render_readme": true,  
"domains": ["sensor", "binary_sensor"],  
"homeassistant": "2025.9.0"  
}
```

The `hacs.json` parameters define how HACS presents and handles the repository: `name` sets the display name shown in HACS, `content_in_root` tells HACS whether the integration files are stored directly in the repository root or under `custom_components`, `render_readme` controls whether the `README.md` content is displayed in the HACS interface, `domains` identifies the Home Assistant entity platforms provided by the integration, and `homeassistant` specifies the minimum Home Assistant version expected to support the integration.

HACS improves the user experience by connecting Home Assistant to a GitHub repository. Once the repository is added to HACS, users can download the integration, receive update notifications, and access the repository documentation more easily. For community projects, this is a better distribution model because it reduces the number of manual steps and makes future maintenance more manageable.

Practical example – preparing the Dual Measurement Sensor integration for HACS

The dual measurement sensor integration created earlier in the book is a good candidate for this process. I published it already in HACS as a custom repository (https://github.com/marcoascarvalho/dual_measurement_sensor) so the next steps will focus on giving clear directions on how to proceed in case you create your own integration. We will have to follow these steps to install as a HACS integration:

1. Create a GitHub repository named `dual_measurement_sensor`.
2. Place the integration files under `custom_components/dual_measurement_sensor`.
3. Add a `README`, `LICENSE`, and `hacs.json` files.
4. Publish the repository on GitHub.
5. Search for the Dual Measurement Sensor integration in HACS.
6. Click on the **Download** button in HACS and follow the instructions. You will have to reboot Home Assistant to install it.
7. Under **Settings | Devices & services | Integrations** | Add Integration, look for the Dual Measurement Sensor integration.
8. Configure the parameters of your sensor and click on **Submit**. The integration entities for your sensor will be created.

Figure 11.8 shows how a device using the dual measurement sensor integration looks.

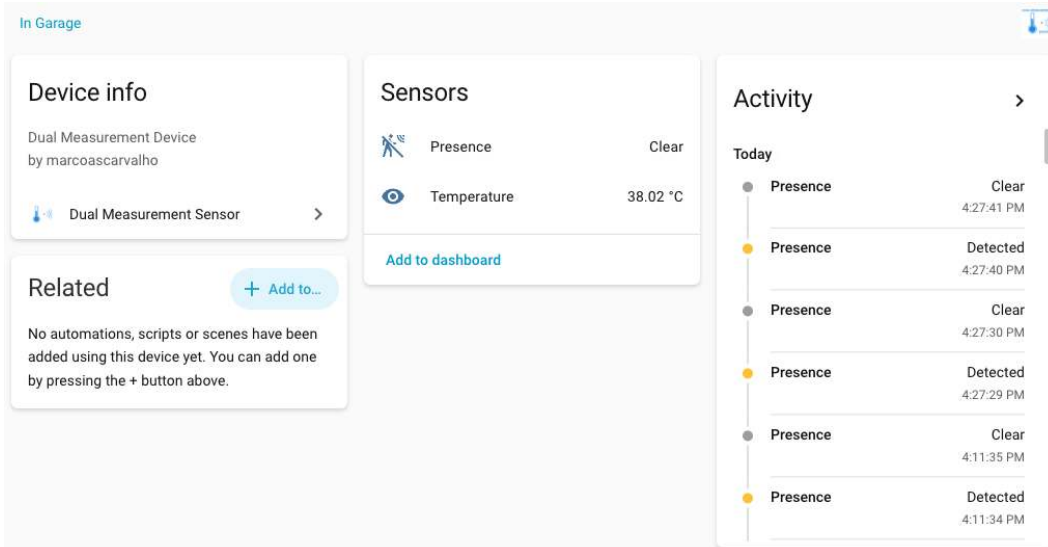


Figure 11.8 – HACS Dual Measurement Sensor integration installed

Understanding the structure of a custom integration helps you see what HACS is actually installing behind the scenes. A HACS integration is still a Home Assistant custom integration, with files such as `manifest.json`, `__init__.py`, `config_flow.py`, `sensor.py`, and `binary_sensor.py`. HACS adds a better distribution method by using GitHub repositories, README documentation, metadata, and update tracking.

The process described gives you a practical path to creating integrations that other Home Assistant users can install, understand, and maintain. The next section focuses on maintaining and updating your HACS ecosystem so that custom integrations, frontend cards, and themes continue to improve your system without compromising stability.

Maintaining and updating your HACS ecosystem

Community content is not part of the official Home Assistant release process, so an update can introduce breaking changes, especially when Home Assistant changes internal APIs or frontend behavior.

For this reason, HACS maintenance should be treated as part of your normal Home Assistant administration routine. This is even more important when a custom integration is used by security automations, dashboards, MQTT devices, energy monitoring, or critical notifications. A failed update may not only remove an entity from a dashboard; it may also break automations that depend on that entity.

Recommended HACS update workflow

Use the following process when you update HACS content:

1. Read the release notes.
2. Check the project's GitHub issues for reported problems.
3. Create a Home Assistant backup.
4. Update one integration, card, or theme at a time.
5. Restart Home Assistant if required.
6. Check the logs.
7. Confirm that entities, dashboards, and automations still work.
8. Update your documentation if configuration changed.

Avoid updating all HACS integrations at once in a production smart home system. If several items are updated together and something breaks, it becomes much harder to identify which update caused the problem.

HACS maintenance checklist and what to verify after an update

A reliable HACS ecosystem depends on regular review. *Table 11.5* presents a checklist that can be used as a simple maintenance routine.

Task	Frequency
Review pending HACS updates	Weekly or biweekly
Read release notes before updating	Every update
Backup before major updates	Every major update
Remove unused HACS integrations	Monthly
Check unavailable entities	Weekly
Review logs after updates	Every update
Test critical automations	After any major change

Table 11.5 – HACS maintenance tasks frequency

After an update, do not assume that the integration is working only because Home Assistant restarted successfully. Validate the main parts of the system affected by the update. *Table 11.6* lists what to verify depending on the Home Assistant topic.

Home Assistant topic	What to verify
Logs	No setup errors, import errors, or repeated warnings
Entities	Expected entities are available and updating
Dashboards	Custom cards still load correctly
Automations	Automations using the updated content still run
Devices	Physical devices still respond as expected

Table 11.6 – What Home Assistant topics to verify after an update

As mentioned before, because HACS updates can cause problems, avoid making several changes at once. First, check the Home Assistant logs and the project's GitHub issues. If the issue is known, you may decide to wait for a fix or roll back to an earlier version if one is available.

If the update affects critical automations or makes the system unstable, restore the backup created before the update.

HACS is one of the most useful ways to extend Home Assistant, but it should be maintained with care. By implementing the maintenance routine proposed here, you can benefit from the flexibility of community integrations while keeping your smart home stable and recoverable.

Summary

In this chapter, you learned how HACS extends Home Assistant by providing a structured way to discover, install, update, and manage community integrations, dashboard cards, themes, templates, and AppDaemon apps.

You reviewed the process for installing, configuring, validating, and troubleshooting HACS content. The chapter explained that different HACS content types require different follow-up steps.

You also explored advanced HACS categories, including dashboard cards, monitoring integrations, device-specific integrations, and themes. These examples showed how HACS can improve dashboards, system reliability, device support, and usability in larger Home Assistant installations.

Finally, you learned how a custom integration, such as the Dual Measurement Sensor, can be prepared for HACS distribution. The chapter closed with a safe maintenance workflow. In *Chapter 12, AI-Driven Home Automation*, you will use Home Assistant data to create smarter and more proactive automations.

12

AI-Driven Home Automation

In the previous chapters, we expanded Home Assistant using advanced automations, Node-RED, AppDaemon, and HACS integrations. These tools allow your smart home to respond to events, execute complex logic, and connect with services beyond the official ecosystem. In this final chapter, we will take the next step by using artificial intelligence to help your smart home understand behavior, detect unusual patterns, and suggest better actions.

Traditional automations are usually based on fixed rules. For example, if motion is detected in the hallway after sunset, the lights turn on. This works well when the expected behavior is clear. However, homes are dynamic environments. People change routines, appliances behave differently over time, rooms are occupied at different hours, and energy usage changes with seasons. AI and machine learning can help identify these patterns and transform historical data into useful recommendations.

This chapter focuses on building a practical and safe AI-assisted workflow for Home Assistant. You will learn how to use Home Assistant data to understand habits, create rule-based suggestions, identify anomalies, and use external AI tools such as ChatGPT, Claude, Gemini, or local LLMs (Large Language Models) to explain data and recommend improvements.

By the end of this chapter, you will be able to apply AI and ML concepts to home automation, use Home Assistant historical data for habit learning and pattern recognition, build recommendation sensors and suggestion workflows, detect unusual events, and integrate AI services, local LLMs, and HACS tools.

In this chapter, the following content will be explored:

- AI and ML in home automation: concepts and possibilities
- Learning consumption habits from Home Assistant data
- Implementing rule-based and simple ML-driven suggestions

- Anomaly detection for security and efficiency
- Integrating external ML services for advanced AI capabilities

Technical requirements

To follow the examples in this chapter, you will need:

- A working Home Assistant installation with historical data enabled
- The Recorder, History, and Statistics features available in Home Assistant
- A few days or weeks of sensor history already stored in your system
- Basic experience with YAML, templates, helpers, and automations, as covered in previous chapters
- Optional experience with Node-RED, AppDaemon, or Python for more advanced data processing examples
- One AI-assisted option, such as:
 - A cloud AI service, such as OpenAI, Anthropic Claude, or Google Gemini
 - A local LLM running through a service such as Ollama
 - A HACS integration that adds AI capabilities, such as camera or event analysis

You do not need all of these tools to complete the chapter. The examples are designed so you can start with native Home Assistant capabilities and then expand into AI services when needed.

AI and ML in home automation: concepts and possibilities

Artificial intelligence, or AI, is a broad term used to describe systems that perform tasks normally associated with human reasoning. These tasks can include recognizing patterns, interpreting language, making predictions, identifying unusual behavior, or generating recommendations. In a smart home, AI can help transform raw sensor data into useful information that supports comfort, efficiency, security, and convenience.

Machine learning, or ML, is a subset of AI. Instead of creating a fixed rule for every possible situation, a machine learning model uses data to identify patterns. For example, rather than manually defining every possible appliance behavior, a model can learn what normal refrigerator, washing machine, or office power usage looks like and then detect when something is different.

In a Home Assistant environment, AI does not need to be complicated to be useful. Many practical use cases begin with simple questions:

- When is this room usually occupied?
- Is this appliance running longer than normal?
- Is today's energy usage higher than expected?
- Did the garage door open at an unusual time?
- Should the thermostat schedule be adjusted based on recent behavior?
- Can the system summarize what happened overnight?
- Can the system suggest automations based on repeated manual actions?

These questions are valuable because Home Assistant already collects a large amount of contextual information. It stores state changes, sensor readings, device activity, energy usage, and automation events. This historical information becomes the foundation for habit learning and pattern recognition.

For example, a motion sensor can tell you that a room is occupied right now. However, historical motion data can show when the room is usually occupied during weekdays, weekends, mornings, and evenings. A smart plug can tell you the current power consumption of an appliance, but historical power data can help identify normal operating cycles, standby consumption, and abnormal runtime.

This is where AI-driven home automation becomes different from traditional automation. A traditional automation reacts to a known condition. An AI-assisted system looks at past behavior and helps identify what may happen next, what changed, or what could be improved.

Figure 12.1 gives an idea how AI can support Home Assistant automations.

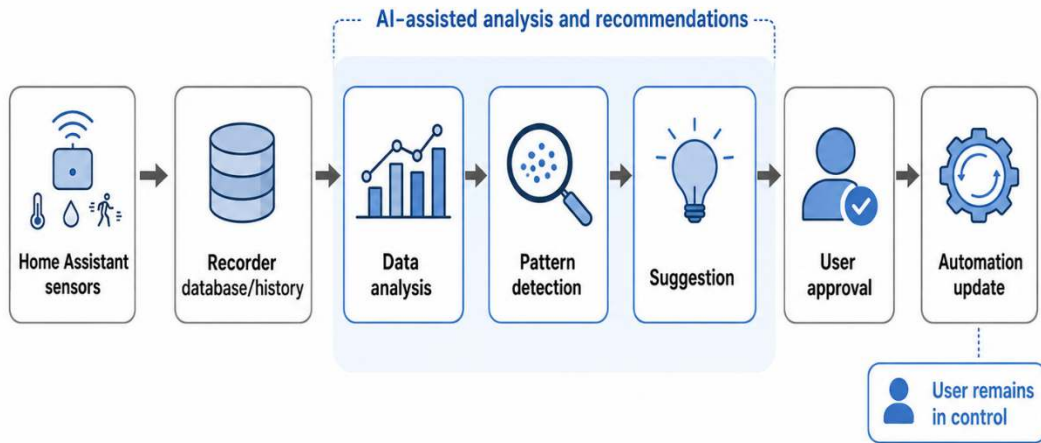


Figure 12.1 – From sensor data to AI suggestions

Figure 12.1 shows the idea proposed by AI-driven Automations. It is not to replace Home Assistant automations but to add an analysis and suggestion layer on top of the data already collected by the system. The end user still remains in control of what should be done.

Rules, Statistics, ML and LLMs

Before creating an AI-driven smart home system, it is important to separate four concepts that are often mixed together: rule-based automation, statistical analysis, machine learning, and large language models.

Table 12.1 defines what these concepts are and provides an example associated to Home Assistant.

Concept	Definition	Example
Rule-based automation	Logic explicitly defined by the user	If motion is detected after sunset, turn on the light
Statistical analysis	Calculations based on historical values	Average refrigerator power during the last 7 days
Machine learning	Model trained to recognize patterns or anomalies	Detect appliance behavior that differs from normal operation

Concept	Definition	Example
Large language model	AI model that understands and generates natural language	Explain why energy usage was high yesterday

Table 12.1 – Definitions and examples of AI-related concepts in Home Assistant

A practical AI-enabled smart home implementation combines rules, statistics, machine learning, and LLMs. Rules remain the best option for predictable actions, such as turning on a light when motion is detected. Statistics help add context by comparing current values with historical behavior, while machine learning can identify patterns or anomalies that are harder to define manually. LLMs add another layer by explaining results, summarizing events, and suggesting new automations in natural language.

With these concepts separated, the next step is to understand where this intelligence should run: locally inside the home, in the cloud, or through a hybrid approach.

Edge AI versus cloud AI

When discussing AI in home automation, it is important to understand that the best processing location depends on the type of intelligence being used. Simple rules and statistics can easily run inside Home Assistant. Machine learning models can run locally when they are small enough or when the hardware is powerful enough. Large language models, or LLMs, usually require more processing power and memory, so they are normally executed in the cloud or on a local server.

Edge AI means that the processing happens close to the device or inside the home network instead of sending all data to an external service. This approach makes sense for tasks such as motion classification, sound detection, image recognition, vibration analysis, and other cases where a small model can make a fast decision locally. In the context of a smart home, it makes sense to split edge AI into the following:

- Device edge: AI or ML runs directly on the end device itself, such as an ESP32, camera, motion sensor, or smart sensor node.
- Home edge: AI processing runs inside the home network in the Home Automation server.

TinyML is one approach used to implement AI at the device edge. It enables small machine learning models to run directly on resource-constrained microcontrollers, including devices based on ESP32-class hardware. Frameworks such as LiteRT for Microcontrollers (formerly TensorFlow Lite for Microcontrollers) can be used to deploy these models for applications such as sensor classification, anomaly detection, gesture recognition, and audio analysis. More information about TinyML is available from the TinyML Foundation (<https://tinyml.org/>),

while implementation information for microcontrollers is available in the Google AI Edge LiteRT documentation (<https://developers.google.com/edge/litert/microcontrollers/overview>).

Platforms such as **Edge Impulse** provide additional tools for collecting sensor data, training and optimizing machine learning models, and deploying them to supported embedded devices. TinyML, LiteRT and Edge Impulse are approaches that allow sensor data to be processed locally without continuously sending raw data to the cloud.

Cloud AI refers to artificial intelligence services that run on external servers instead of inside the smart home network or directly on local devices. In this approach, Home Assistant or another local system sends selected data, usually summarized or filtered, to an online AI service for processing.

The trend is that smart home AI implementations use a hybrid AI architecture, combining edge AI for local sensing, automation, and critical control with cloud AI or local LLMs for summaries, explanations, and recommendations. *Figure 12.2* shows this balance, where essential decisions remain local while selected and summarized data can be sent to AI services when deeper analysis is useful.

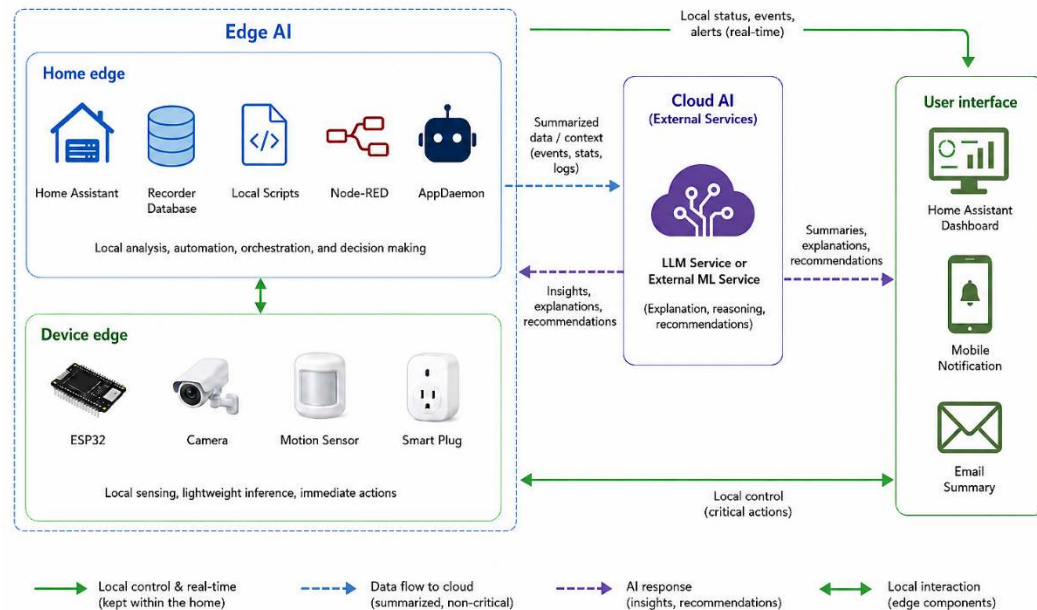


Figure 12.2 – Edge and cloud AI in a hybrid AI architecture

Figure 12.2 shows how a practical smart home AI architecture can combine device edge, home edge, and cloud AI. Device edge components handle local sensing and immediate actions,

while the home edge layer is used for local analysis and orchestration. Cloud AI or external ML services can then receive selected summarized data to generate explanations, insights, or recommendations, while critical control remains inside the home network.

What AI should and should not control in your home

AI can be very useful, but it should not be given unlimited control over a home. A safe strategy is to separate suggestions from actions. This is especially important when the automation affects safety, security, access control, heating, cooling, appliances, or electrical loads.

For example, it is safer for AI to generate this recommendation:

"During the last 10 weekdays, the washing machine was started between 5:00 PM and 7:00 PM, which matches a higher energy-cost period. Consider creating an automation to notify the user when the washing machine starts during peak hours and suggest running it later, such as after 9:00 PM or during another lower-cost energy period."

A practical recommended AI-driven workflow is the following sequence presented in *Figure 12.3*.

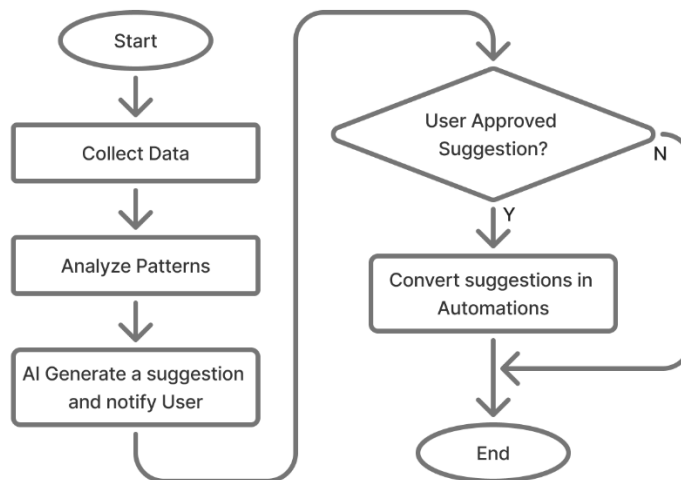


Figure 12.3 – AI-driven suggested workflow

This model presented in *Figure 12.3* keeps the user in control. AI becomes an assistant that observes, explains, and recommends instead of an uncontrolled decision maker. This is a better fit for smart homes because the consequences of a wrong action can affect comfort, energy cost, security, or safety.

Practical possibilities for AI in smart homes

AI-driven home automation can be applied to many areas. Here are some examples of where it can be used:

- **Energy usage optimization:** Home Assistant can collect power readings from smart plugs, energy meters, solar inverters, thermostats, and HVAC systems. By analyzing this data, the system can identify high-consumption periods, unusual appliance behavior, or opportunities to shift usage to better times. We will explore an example in the next section using this.
- **Occupancy learning:** the system can learn when rooms are usually occupied. This can help improve lighting schedules, climate control, and notifications.
- **Security:** A rule-based system can detect a door opening at night, but AI-assisted analysis can compare the event with normal behavior. For example, opening the garage at 6:30 PM may be normal, while opening it at 2:45 AM may require a different response.

AI can also help with user experience. Instead of making the user inspect logs, dashboards, and history graphs manually, an AI assistant can summarize key events in natural language:

"Last night, the garage door remained closed, no motion was detected after 11:30 PM, and the HVAC system ran longer than usual between 3:00 AM and 4:00 AM."

As seen in this section, AI should complement Home Assistant, not replace its automation engine. It adds a layer for interpreting data, finding patterns, detecting anomalies, and explaining behavior. By combining rules, statistics, machine learning, and language models in a controlled way, the smart home can generate useful suggestions. In the next section, Home Assistant data will be used to learn consumption habits and identify patterns that support these suggestions.

Learning consumption habits from Home Assistant data

Home Assistant already stores a valuable record of how your home behaves. This data can include motion events, door openings, temperature changes, humidity readings, power consumption, light usage, media player activity, thermostat operation, and presence information. When organized correctly, this historical data can reveal daily routines, repeated manual actions, appliance usage habits, comfort preferences, and unusual behavior.

The first step in AI-driven automation is to select the right data. A smart home cannot learn meaningful habits if the data is noisy, incomplete, or unrelated to the behavior you want to understand. Before creating suggestions, you need to decide which entities represent useful

activity, and which entities should be filtered, ignored, or converted into simpler derived values.

The complete workflow used in this section is presented in *Figure 12.4*. Home Assistant first collects sensor data and stores historical states in Recorder. The data is then processed into meaningful usage sessions and analyzed by an AI Task to identify habits and generate recommendations. The AI results are stored in helpers and exposed through shadow sensors, which can trigger notifications and recommendations. This workflow provides a roadmap for the implementation developed throughout the following subsections.

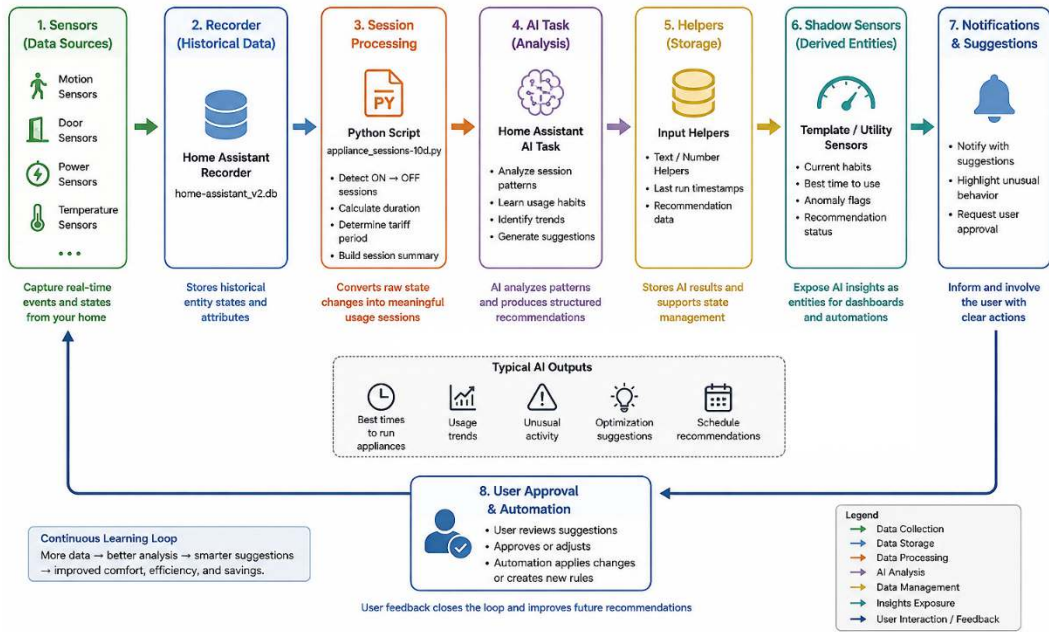


Figure 12.4 – AI-driven automation and continuous learning workflow in Home Assistant

Data sources and quality

The best entities for habit learning are usually the ones that describe repeated behavior. These include sensors that show when a room is occupied, when an appliance is used, when a door is opened, or when a device is manually controlled. These events can then be transformed into features such as runtime, number of activations, first event time, last event time, average value, or total daily usage.

Table 12.2 shows some examples of Home Assistant data sources for habit learning.

Data source	What it can reveal	Example suggestion
Motion sensors	Room occupancy trends	Suggest adjusting light schedules
Door sensors	Entry and exit patterns	Notify if a door opens at unusual hours
Power sensors	Appliance usage habits	Suggest off-peak appliance operation
Temperature sensors	Comfort and HVAC behavior	Suggest thermostat schedule changes

Table 12.2 – Useful Home Assistant data sources for habit learning

AI is only as good as the data quality provided to it. You have to verify that your entities are named correctly, sensors are stable, and important data is being recorded. This is where the database optimization techniques from *Chapter 1* become important. If you exclude a sensor from Recorder, it may still work live in Home Assistant, but it will not be available for historical analysis.

You should also avoid analyzing raw noisy data when a cleaner derived entity can be created first. For example, instead of analyzing every power fluctuation from a smart plug, you can create a binary sensor that shows whether the appliance is running. This simplifies the analysis and reduces the chance of generating incorrect suggestions.

An example of a template sensor created from a smart plug with a power sensor, representing whether an appliance is running, is provided below:

```
template:
  - binary_sensor:
      - name: "AirFryer Running"
        unique_id: airfryer_running
        delay_off:
          minutes: 5
        state: >
          {{ states('sensor.airfryer_energy_current') | float(0) > 0 }}
```

This template creates a derived binary sensor named AirFryer Running. The `delay_off` parameter keeps the sensor on for 5 minutes after the current drops to zero, which avoids false off states during short pauses in the appliance cycle. The state template reads the current value from `sensor.airfryer_energy_current`, converts it to a number using `float(0)`, and turns the binary sensor on when the current is greater than 0.

Privacy is another important part of data quality. Home Assistant history can reveal when people are home, when they sleep, when doors are opened, and when cameras or motion sensors detect activity. You should avoid sending raw database exports to cloud AI services. A better approach is to summarize data locally first before sending the summary to an AI service. An example of how to do that will be presented in the next subsection.

Preparing sensor data to learn usage habits

We will start preparing the data to be used by AI to help with habit learning for some power sensors' usage. Four power sensors will be used to implement this example:
`binary_sensor.sandwich_grill_on`, `binary_sensor.dishwasher_running`,
`binary_sensor.airfryer_running` and `binary_sensor.wash_machine_running`.

To learn when a device is normally used, the system needs a session summary. A session starts when a sensor changes to on and ends when it changes back to off. Each session will have data represented as the examples mentioned in *Table 12.3*.

Entity	Start	End	Durati on	Tariff Period
<code>binary_sensor.sandwich_grill_on</code>	2026-05-04 15:22	2026-05-04 15:27	5 min	peak
<code>binary_sensor.dishwasher_running</code>	2026-05-04 21:40	2026-05-04 23:10	90 min	off_peak

Table 12.3 – Power sensor sessions formatted data

Home Assistant by default stores 10 days of entities data. I created a Python program, `appliance_sessions_10d.py`, which reads the Home Assistant database file `home-assistant_v2.db`, builds the sensor sessions for these 10 days, and includes them in the format of *Table 12.3*. The program implements a tariff scheme from the energy company of my location and you should adapt the code to support your energy tariffs. The code is available in the book's GitHub repository. To access the repository link, follow the steps in the "Download the example code files" section in the *Preface*.

Note

Reminder: To store entities data for more than 10 days, adjust the `purge_keep_days` number under the recorder section in `Configuration.yaml`.

This session-based format is much easier for AI to understand than raw state history. It allows the AI model to identify common usage windows, unusual activity, and whether usage normally happens during peak or off-peak energy periods.

The output of the Python program created will be included in a sensor to be read by a Home Assistant automation. To do that, a command line sensor will be created in Home Assistant to run the python program and store the output sessions data in the sensor attributes area. The YAML code that implements the command line sensor is the following:

```
command_line:
  - sensor:
      name: Appliance Usage Sessions 10d
      unique_id: appliance_usage_sessions_10d
      command: "python3 /config/scripts/appliance_sessions_10d.py"
      command_timeout: 60
      scan_interval: 604800
      value_template: "{{ value_json.state }}"
      json_attributes:
        - generated_at
        - days
        - session_count
        - summary
```

This `command_line` sensor creates a Home Assistant sensor named **Appliance Usage Sessions 10d** by running the Python script located at `/config/scripts/appliance_sessions_10d.py`. The `command_timeout: 60` gives the script up to 60 seconds to finish, while `scan_interval: 604800` runs it once every 604,800 seconds, or once per week. The `value_template` sets the sensor state from the JSON field `state`, such as `ready`. The `json_attributes` list stores extra JSON fields from the script output as sensor attributes, including when the summary was generated, how many days were analyzed, how many sessions were found, and the AI-ready summary text.

Because the sensor attribute may contain a long text summary, it is a good practice to exclude this sensor from Recorder to prevent Home Assistant from repeatedly storing the generated summary inside its own database. This is done using the following YAML code:

```
recorder:
  exclude:
    entities:
      - sensor.appliance_usage_sessions_10d
```

An AI Task to learn the habits from the sensor sessions will be presented in the next section.

Using AI Task in Home Assistant

AI Task became available as part of Home Assistant's 2025.8 AI-focused release (<https://www.home-assistant.io/blog/2025/08/06/release-20258/>). It provides the `ai_task.generate_data` action, which allows scripts and automations to send instructions to a configured AI provider and receive text or structured data in return. To use it you will have to follow these steps:

1. Install and configure an AI provider integration, such as OpenAI, Google Gemini, Anthropic, or Ollama.
2. Add or enable an AI Task entity for that provider.
3. Use the `ai_task.generate_data` action in a script or automation.
4. Provide clear instructions and, when possible, structured output fields.
5. Capture the response with `response_variable`.
6. Use the response to send a notification, update helpers, or drive recommendation logic.

The AI Task action will not query Home Assistant history by itself. It analyzes the data provided in the instructions. For this reason, the session summary sensor created earlier becomes the bridge between Home Assistant history and AI analysis.

A simplified AI task script pseudocode can look like the example below:

```
alias: Analyze Appliance Trends with AI
mode: single
sequence:
  - action: homeassistant.update_entity
    target:
      entity_id: sensor.appliance_usage_sessions_10d
  - delay:
      seconds: 5
  - action: ai_task.generate_data
    response_variable: appliance_ai_result
    data:
      entity_id: ai_task.openai_ai_task
      task_name: appliance_usage_trend_analysis
      instructions: |-
        ...
        AI PROMPT HERE
        ...
```

The `entity_id` in this action should be replaced with the actual AI Task entity configured in your system, such as `ai_task.openai_ai_task`.

You will have to decide which AI services you will use. In my case I decided to use OpenAI (<https://platform.openai.com/login>). Before installing the OpenAI integration, you will have to create a Long-lived access token using Profile | Security and create an API key in the OpenAI platform login page. After it, install the OpenAI integration using Settings | Devices & Services | OpenAI. The integration installed will show as represented in *Figure 12.5*.

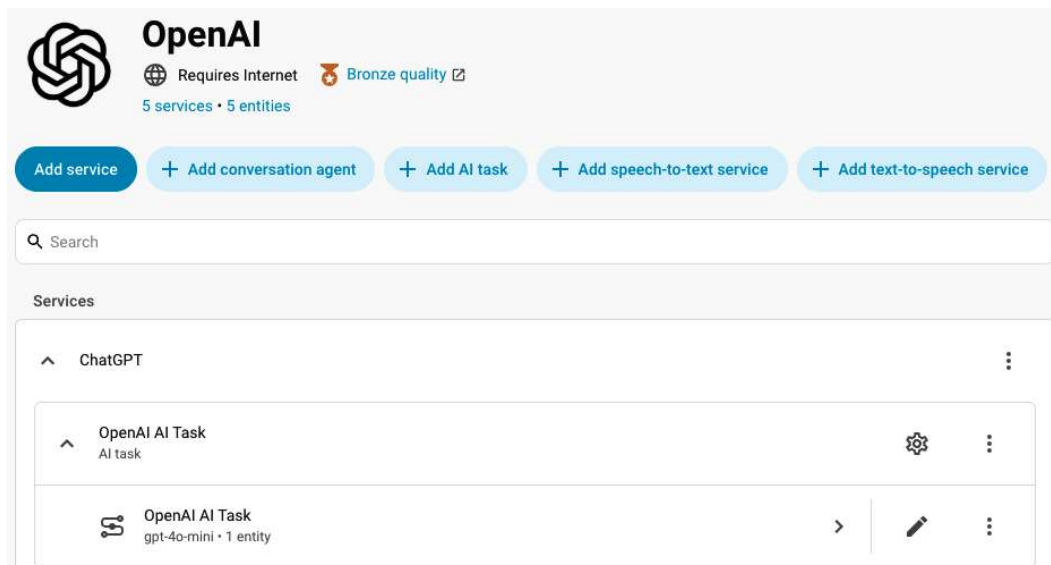


Figure 12.5 – OpenAI integration installed

If you decide to use Google Gemini, Anthropic Claude, Ollama, or another provider, the entity name will be different, and you will have to configure the correspondent integration.

The AI PROMPT HERE in the previous YAML pseudocode is where you will include the AI instructions on what you want it to do. For our example in question, we can use something like the following prompt:

```
You are analyzing Home Assistant appliance usage sessions.

Appliance session summary: {{ state_attr('sensor.appliance_usage_sessions_10d',
'summary') }}
```

Based only on the supplied data, return learned expected usage windows and recommended usage windows.

Definitions:

- expected_start and expected_end represent the normal learned usage window.
- recommended_start and recommended_end represent the preferred usage window considering my local tariffs and practical use.
- Use 24-hour HH:MM format only, such as 18:00 or 21:30.
- Do not include AM, PM, seconds, dates, or explanatory text in time fields.
- If there is not enough data for an appliance, return 00:00 for start and 00:00 for end and set confidence to low.

Return time windows for:

- dishwasher
- washing_machine
- airfryer
- sandwich_grill

As part of the AI task configuration, a structured output can be used to return the answers in predictable fields. The AI Task can return named values such as `dishwasher_expected_start`, `dishwasher_expected_end`, `dishwasher_recommended_start` and `dishwasher_recommended_end`.

The structure block of the AI task can look like the following code:

```
structure:
  dishwasher_expected_start:
    description: Dishwasher expected usage start time in 24-hour HH:MM format.
    required: true
    selector:
      text:
    ...

  summary:
    description: Short explanation of the appliance trend analysis and tariff
    optimization recommendations.
    required: true
    selector:
      text:
```

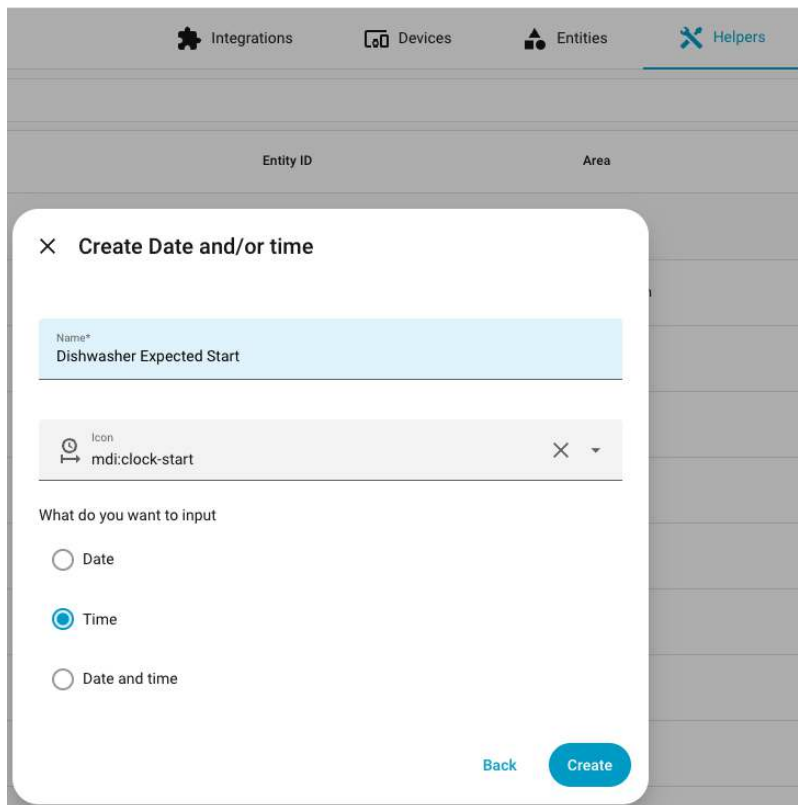
The full example implementation should include the same set of fields for the washing machine, air fryer, and sandwich grill. Using the structured output option makes the AI result easier to use in later actions because each value can be referenced directly. For example,

dishwasher_expected_start, can be accessed using
`{{ appliance_ai_result.data.dishwasher_expected_start }}`.

Updating helpers from AI results

After the AI Task generates structured data, Home Assistant can update helper entities. These helpers store the latest learned and recommended time windows. For each interested sensor, you will have to create four helpers to store the expected start time, expected end time, recommended start time and recommended end time.

To create a helper, you will have to go to Settings | Devices & services | Helpers | Create helper button and choose the option Date and/or time. Fill in the information according to the screen presented in *Figure 12.6* for each sensor including the Name textbox and Time checkbox at least.



The screenshot shows the 'Create Date and/or time' dialog in the Home Assistant interface. The dialog is overlaid on a background showing the 'Helpers' tab in the settings menu. The dialog has a title bar with a close button (X) and the title 'Create Date and/or time'. Below the title bar, there is a 'Name*' field with the text 'Dishwasher Expected Start'. Below the name field, there is an 'Icon' field with a clock icon and the text 'mdi:clock-start'. Below the icon field, there is a section titled 'What do you want to input' with three radio button options: 'Date', 'Time' (which is selected), and 'Date and time'. At the bottom right of the dialog, there are two buttons: 'Back' and 'Create'.

Figure 12.6 – Creating time helpers

The AI script `Analyze Appliance Trends` with AI continuation can now add actions to update each one of the helpers created. The YAML code to include the action to update the helper created on *Figure 12.6* is the following.

```
- action: input_datetime.set_datetime
  target:
    entity_id: input_datetime.dishwasher_expected_start
  data:
    time: "{{ appliance_ai_result.data.dishwasher_expected_start }}:00"
```

Repeat this action for the `expected_end`, `recommended_start` and `recommended_end` helpers, for as many sensors as you want to track. In the case of this example, I did it for my washing machine, sandwich grill, and air fryer.

This concludes the implementation of the AI task script started in the previous section. In the next section, the helpers will be used to update shadow entities.

Creating shadow entities

The helpers created in the previous subsection can now drive shadow binary sensors. A shadow entity represents an expected or recommended condition. They are defined as:

- Expected shadow sensor: represents the normal usage window discovered from historical Home Assistant data, such as when the dishwasher is usually used.
- Recommended shadow sensor: represents the preferred usage window after applying additional logic, such as off-peak energy tariffs, safety, or convenience.

The YAML code for the dishwasher Expected shadow sensor is the following:

```
- name: Dishwasher Expected Usage Time
  unique_id: dishwasher_expected_usage_time
  icon: mdi:dishwasher
  state: >
    {% set start = states('input_datetime.dishwasher_expected_start') %}
    {% set end = states('input_datetime.dishwasher_expected_end') %}
    {% set now_min = now().hour * 60 + now().minute %}
    {% set s = start.split(':') %}
    {% set e = end.split(':') %}
    {% set start_min = s[0] | int * 60 + s[1] | int %}
    {% set end_min = e[0] | int * 60 + e[1] | int %}
    {% if start_min <= end_min %}
      {{ start_min <= now_min < end_min }}
    {% else %}
```

```

    {{ now_min >= start_min or now_min < end_min }}
    {% endif %}

```

This template creates a shadow binary sensor named `Dishwasher Expected Usage Time`. It reads the expected start and end times from the `input_datetime.dishwasher_expected_start` and `input_datetime.dishwasher_expected_end` helpers, converts those times and the current time into minutes, and then checks whether the current time falls inside the expected usage window. The final `if` logic supports both normal windows, such as 18:00 to 21:00, and overnight windows, such as 21:00 to 06:00. When the current time is inside the learned dishwasher usage window, the sensor turns on, otherwise, it stays off.

Once the shadow sensors are created, it is time for the automation based on them. The next section will show how to do it.

Sending notifications from shadow sensors and running periodic AI trend analysis

Once the shadow sensors exist, Home Assistant can notify the user when an appliance starts outside an expected or recommended window. The following example automation can be created to recommend a usage time.

```

alias: Dishwasher Started Outside Recommended Usage Time
triggers:
  - trigger: state
    entity_id: binary_sensor.dishwasher_running
    to: "on"
conditions:
  - condition: state
    entity_id: binary_sensor.dishwasher_recommended_usage_time
    state: "off"
actions:
  - action: notify.mobile_app_iphone_381
    data:
      title: Dishwasher energy suggestion
      message: >
        The dishwasher started outside its recommended usage window.
        Recommended window:
        {{ states('input_datetime.dishwasher_recommended_start') }} to
        {{ states('input_datetime.dishwasher_recommended_end') }}.
mode: single

```


For a meal-time appliance such as a sandwich grill, the notification may focus on unusual usage instead of energy shifting. So, the message can be something like part of the YAML code below:

```
actions:
  - action: notify.mobile_app_iphone_381
    data:
      title: Sandwich grill unusual usage
      message: >
        The sandwich grill started outside its expected usage window.
        Expected window:
        {{ states('input_datetime.sandwich_grill_expected_start') }} to
        {{ states('input_datetime.sandwich_grill_expected_end') }}.
    mode: single
```

To finalize the entire process of AI trends, we can automate the flow by creating an automation that runs periodically. As standard, the YAML code below will run every 10 days:

```
alias: AI Appliance Trend Review Every 10 Days
triggers:
  - trigger: time
    at: "08:00:00"
conditions:
  - condition: template
    value_template: "{{ now().timetuple().tm_yday % 10 == 0 }}"
actions:
  - action: script.analyze_appliance_trends_with_ai
    mode: single
```

The periodic workflow becomes the sequence:

1. Update the appliance session summary
2. Run AI Task analysis
3. Return structured expected and recommended windows
4. Update helper time values
5. Shadow binary sensors reflect the new windows
6. Notify the user with the latest AI recommendations

This creates a repeatable learning process in which the system does not need to be manually reconfigured every time habits change. Instead, it periodically reviews recent history and

adjusts the learned and recommended usage windows, completing the AI-driven automation loop illustrated in *Figure 12.4*.

Other AI Task examples

The same AI Task concept can be used for other smart home tasks beyond appliance usage.

For example, a doorbell camera can provide an image to an AI Task, and the model can identify what triggered the event. The result could classify whether the image contains a person, a package, a vehicle, or an animal. This can make notifications more useful. The sequence presented in *Figure 12.7*.

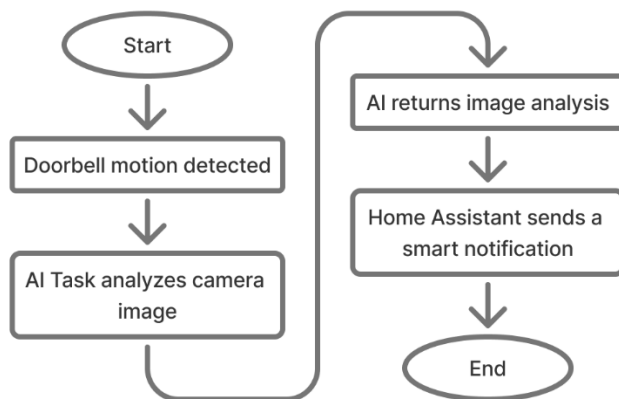


Figure 12.7 – AI Task used for image analysis

Another example is using a camera to count people or animals in an area. A camera in a backyard could detect whether there are people, dogs, cats, or wildlife. A garage or driveway camera could help determine whether a vehicle is present. A coop, patio, or porch camera could count animals or identify whether a delivery was dropped off. Some of these examples were first explored in the YouTube video *Current State of AI task* by Jean-Loïc Pouffier available at: <https://www.youtube.com/watch?v=hf1u0bw7SFY>.

AI image generation tasks can also be used when the goal is to create an image from instructions. This is different from image recognition. Image recognition analyzes an existing media source, while image generation creates a new image based on a prompt. For home automation, image recognition is usually more practical, but image generation may be useful for dashboards, visual summaries, or notification illustrations.

The workflow presented in this section shows how Home Assistant data can move from historical records to learned and recommended behavior. By converting appliance activity into sessions, analyzing those sessions with AI Task, and storing the results in helper-driven

shadow sensors, the system can identify normal usage and communicate preferable usage windows.

Implementing rule-based and simple ML-driven suggestions

This section shows how the collected data can be transformed into actionable suggestions using two different approaches: fixed rules and machine learning. The rule-based method evaluates known conditions, while the machine learning method uses historical data to identify recurring behavior that may not be obvious from a single event.

These approaches solve different problems. A fixed rule can determine whether a known dishwasher cycle occurred during an expensive period, but it cannot determine whether that time represents a recurring habit. Machine learning complements the rule by analyzing many historical records and estimating when the dishwasher is normally used. Together, they allow Home Assistant to evaluate both the latest event and the longer-term behavior.

Rule-based suggestion

Rule-based systems are one of the simplest forms of artificial intelligence because they make decisions by evaluating predefined conditions rather than learning from historical data. They are deterministic, easy to understand, and straightforward to troubleshoot.

In this example, the goal is to create a simple energy-saving recommendation for the dishwasher. Home Assistant will monitor when the dishwasher is used and how much energy it consumes. If a cycle occurs during a higher electricity-rate period, the system will estimate the potential savings from shifting similar usage to a lower-rate period and notify the user when the savings justify changing the schedule. The system does not automatically control the dishwasher; it provides a recommendation that the user can apply to a future cycle.

The rule is straightforward:

"If the dishwasher was used during the higher-rate period, estimate how much could be saved by running it after the higher-rate period ends. If there is a meaningful savings, recommend delaying the next cycle."

The high rate energy period and the cost is simplified by using the below definitions. You should adjust it according to your electricity plan.

```
HIGH_RATE_START_HOUR = 15
HIGH_RATE_END_HOUR = 19
HIGH_RATE_USD_PER_KWH = 0.24
LOW_RATE_USD_PER_KWH = 0.11
```

An automation implements the complete rule-based cycle by coordinating the required sensors, shell command, and Python scripts. The automation flow is presented in *Figure 12.8*.

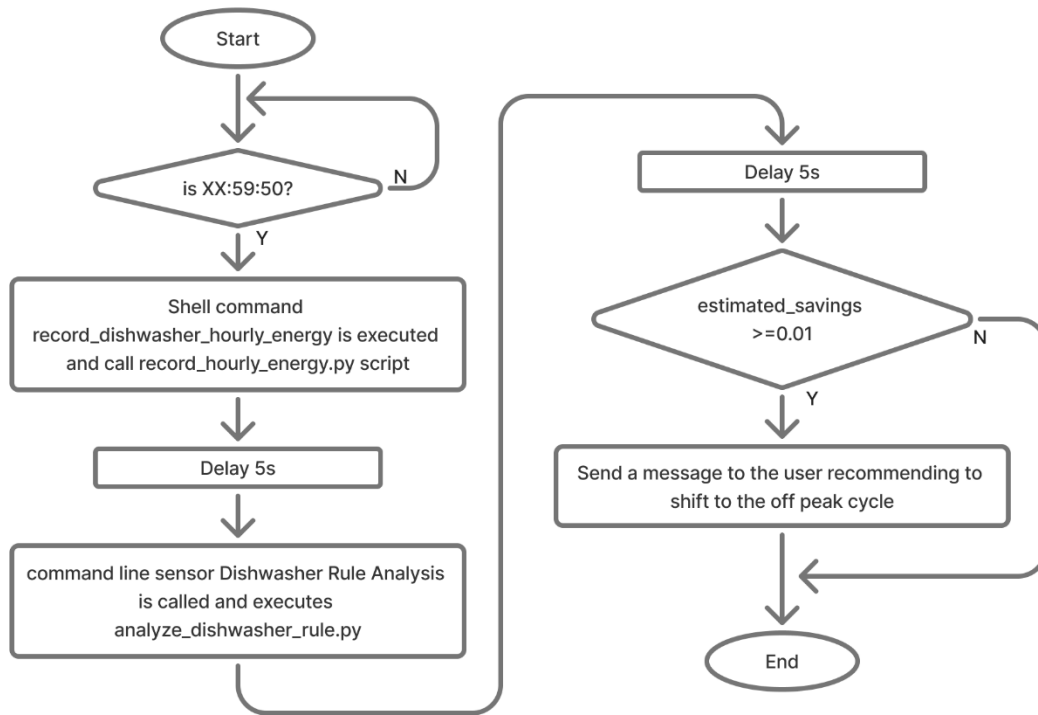


Figure 12.8 – Rule-based automation

To implement this rule-based automation example, I used the following components:

- **Utility Meter:** `sensor.kitchen_dishwasher_hourly_energy`. This sensor is created using Settings | Devices & services | Helpers | Create helper | Utility Meter. The source sensor is `sensor.dishwasher_energy_today` and the meter reset cycle parameter is configured as Hourly.
- **Python script:** `record_dishwasher_hourly_energy.py`. This script stores the hourly energy reading in the CSV file located at: `/config/www/dishwasher_hourly_energy.csv`. Each row contains the timestamp, hour, weekday

indicator, and energy used in the last hour in kWh. The script is called using a shell command `record_dishwasher_hourly_energy`:

```
record_dishwasher_hourly_energy: >
python3 /config/scripts/record_hourly_energy.py
"{{ states('sensor.kitchen_dishwasher_hourly_energy') | float(0) }}"
```

- Python script: `analyze_dishwasher_rule.py`. Reads the most recent meaningful row from the CSV file and applies the rule:

If the usage occurred during the higher-rate period, calculate the potential savings from running the dishwasher after the lower-rate period begins. If the usage occurred during the lower-rate period, no shift is recommended.

- Command Line Sensor: Dishwasher rule Analysis. This sensor runs `analyze_dishwasher_rule.py` and exposes the result to Home Assistant. The sensor state output is either:
 - **Recommended:** shifting the dishwasher cycle would likely save money.
 - **Not_Recommended:** no schedule shift needed.
 - **No_data:** No meaningful dishwasher usage is available.
 - **Error:** The analysis could not be completed.

In `analyze_dishwasher_rule.py`, the usage period is identified by creating a Boolean variable:

```
used_during_high_rate = (HIGH_RATE_START_HOUR <= usage_hour < HIGH_RATE_END_HOUR)
```

The estimated savings are calculated using:

```
current_cost = (energy_kwh * current_rate)
lower_rate_cost = (energy_kwh * LOW_RATE_USD_PER_KWH)
estimated_savings = max(current_cost - lower_rate_cost, 0.0,)
```

After the command-line sensor is refreshed, the automation waits briefly and verifies that the sensor state is recommended and that the estimated savings are at least \$0.01. If both conditions are true, Home Assistant sends the recommendation to the user using the reason attribute generated by the analysis script.

The recommendations using a machine learning implementation will be refined in the next session.

Implementing a simple machine learning recommendation

The rule-based example evaluates only the most recent dishwasher cycle. While this works well when the decision criteria are already known, it cannot determine whether the observed behavior represents a recurring habit. This is where machine learning becomes valuable. Instead of relying on predefined rules, a machine learning model analyzes historical data to discover patterns and make predictions automatically.

In this example, we use the same `dishwasher_hourly_energy.csv` file created by the previous automation. Rather than evaluating only the latest dishwasher cycle, the model analyzes all recorded usage and predicts the hour when the dishwasher is most likely to be used.

The goal is to determine whether the household has developed a recurring dishwasher usage habit that overlaps with the higher-rate electricity period. If it does, the system can recommend shifting future dishwasher cycles to a lower-rate period.

To keep the implementation simple, I used a k-nearest-neighbors (KNN) model (https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm). Unlike many machine learning algorithms, KNN does not require a separate training step or a saved model file. Each time the script executes, it loads the historical records from `dishwasher_hourly_energy.csv` and uses them directly to make a prediction. This type of algorithm is often referred to as a lazy-learning model because the historical data itself serves as the model.

The automation `AI - Analyze Dishwasher Usage Habit` organizes the execution sequence of the example. This automation flow is presented in *Figure 12.9*.

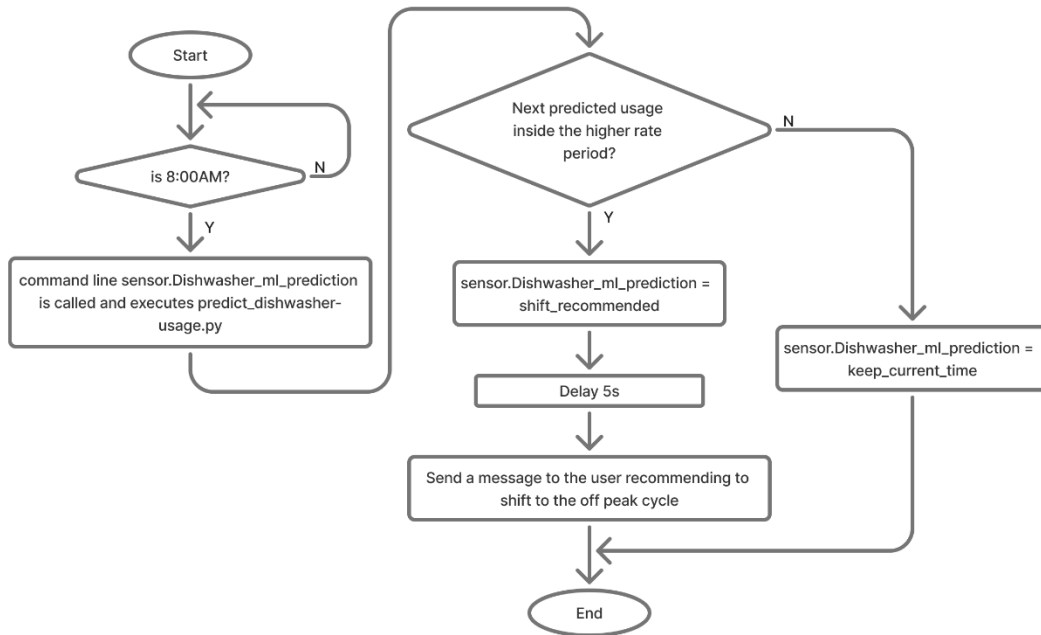


Figure 12.9 – Machine learning example flow

The prediction script, `predict_dishwasher_usage.py`, loads the historical dishwasher records and compares them using the hour of the day, the day of the week, and whether the day is a weekday or weekend. It predicts the expected dishwasher energy consumption for each hour of the current day and selects the hour with the highest predicted energy as the learned dishwasher usage time.

The K-nearest-neighbors algorithm is implemented primarily by the `predict_energy()` function. It calculates the distance between the target hour and every historical record, selects the seven nearest neighbors, and computes a weighted average of their energy consumption. The `calculate_distance()` function defines how similarity is measured, while the `main()` function repeatedly applies the KNN algorithm to all 24 hours of the day and selects the hour with the highest predicted energy consumption as the learned dishwasher usage time.

To ensure that a prediction is not generated from too little data, the script first verifies that at least 48 hourly records and three dishwasher usage events are available. These values are sufficient for demonstrating the workflow, although several weeks of historical data will normally produce a more representative result.

Once enough data has been collected, the script updates the command-line sensor `sensor.dishwasher_ml_prediction`. This sensor exposes the predicted usage time, confidence information, and a recommendation. If the learned schedule falls within the configured higher-rate period, Home Assistant can notify that shifting the dishwasher cycle to a later time could reduce electricity costs.

The same workflow for rule-based and machine learning approaches can be applied to other appliances by creating a utility meter, CSV file, command-line sensor, and automations for each device. The Python scripts can be duplicated or generalized to accept the appliance name and configuration as inputs.

This example illustrates the complementary roles of rule-based logic and machine learning. The rule-based approach determines whether a known dishwasher cycle was expensive, while the machine learning model discovers when the dishwasher is normally used. Combining both techniques produces a more useful recommendation than either approach alone.

Anomaly detection for security and efficiency

In the previous sections, we used AI to learn normal behavior and generate recommendations based on historical data. Another valuable AI capability is recognizing when something differs from those learned patterns. This process is known as anomaly detection.

An anomaly is simply an event that deviates from normal operation. It does not necessarily indicate a failure, but it deserves additional attention. In your home, anomalies may reveal equipment problems, energy waste, security concerns, or sensor failures before they become more serious issues.

Many anomaly detection techniques can be implemented locally using Home Assistant's native features. AI becomes valuable after the anomaly is detected by helping interpret the event, correlate multiple sensors, and generate clear, contextual explanations.

Common smart home anomalies

Almost every smart home already contains enough information to detect unusual behavior.

None of the anomaly situations automatically indicate a problem. A late-night garage door opening may simply mean someone returned home later than usual, while longer HVAC runtimes may result from unusually hot weather. The goal is to identify situations that deserve additional investigation.

Table 12.4 shows some example anomalies and possible causes.

Category	Example	Possible Cause
Energy	Refrigerator running continuously	Door open, dirty coils, compressor problem
Security	Garage door opens while Away	Family member or unauthorized access
Occupancy	Motion detected at unusual hours	Visitor, pet, or unexpected activity
Sensors	Temperature = -40°C	Sensor disconnected
Connectivity	Zigbee device unavailable	Weak mesh or battery issue

Table 12.4 – Example anomalies and possible causes

AI as an explanation layer

A useful design principle is to separate detection from interpretation.

Home Assistant should perform anomaly detection locally using deterministic rules, thresholds, or historical statistics. These methods are reliable, fast, and continue working even without Internet connectivity.

Artificial intelligence can then analyze the detected event and provide additional context. An LLM can compare the event with previous behavior, explain why it may be abnormal, and suggest possible causes. The anomaly itself can be detected using Home Assistant rules or AI. AI can expand the anomaly detection by transforming structured sensor data into a more meaningful explanation.

As an example, I identified an anomaly investigating the daily energy consumption of my Garage Refrigerator explored in *Chapter 3*. Looking at the history of the `refrigerator_garage_energy_today` sensor entity, I could see an anomaly where the energy consumption was below normal on July 11th and July 12th. The history chart representing this behavior is presented in *Figure 12.10*.

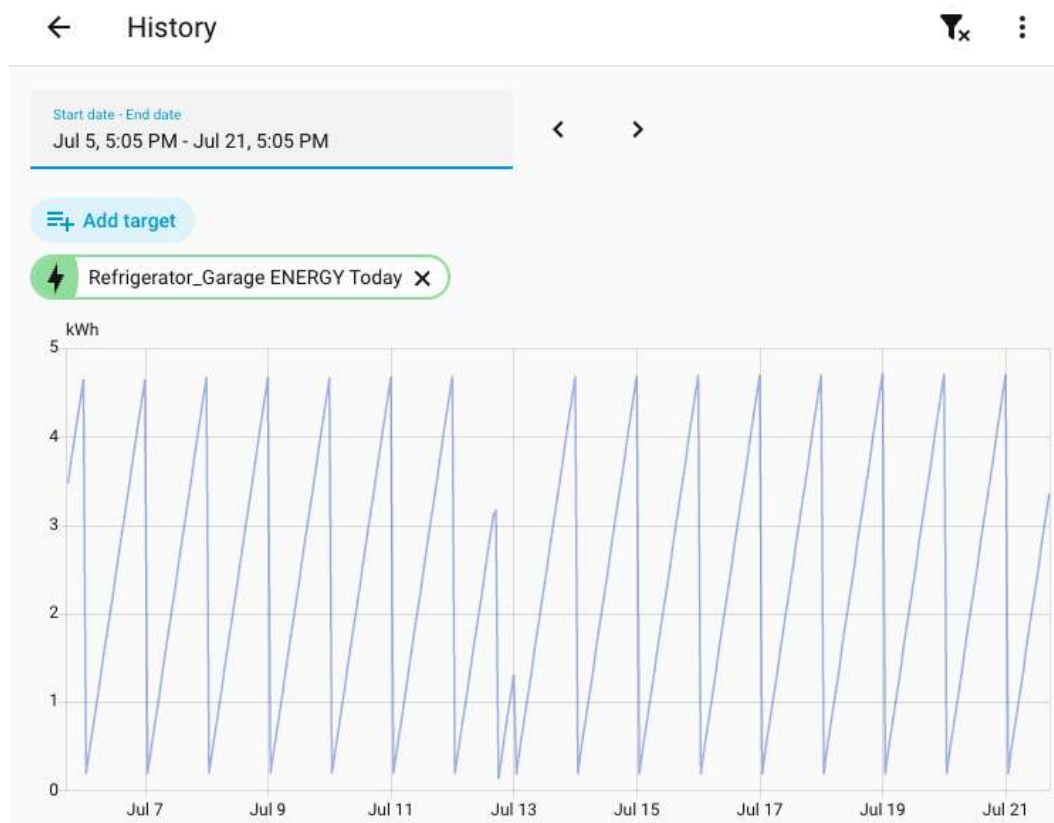


Figure 12.10 – Garage refrigerator anomaly behaviour

Similar to what was presented in the Learning consumption habits from Home Assistant data section, an AI task can be created to verify this sensor on a monthly basis and check for anomalies like the one presented on *Figure 12.10* to provide an explanation. In the case of the sensor `.garage_refrigerator`, the reason for the anomaly was due to an outage that happened due to a weather condition.

To detect anomalies accurately, the AI needs historical data for the main sensor and related sensors, such as garage and outdoor temperature, humidity, motion, and garage door activity. Information about Home Assistant restarts, sensor availability, and weather also helps the AI determine whether unusual behavior was caused by normal conditions, a data interruption, or a possible equipment problem.

Best practices in anomaly detection

When designing AI-assisted anomaly detection systems, follow these guidelines:

- Detect anomalies locally using deterministic logic whenever possible.
- Combine multiple sensors to provide context and reduce false positives.
- Use AI to explain anomalies rather than detect them.
- Include historical comparisons in AI-generated summaries.
- Monitor sensor reliability to ensure trustworthy AI recommendations.

By combining Home Assistant's native automation capabilities with AI-generated explanations, you can build a smart home that helps you understand what these events mean. This hybrid approach improves security, energy efficiency, and system reliability while keeping the detection process fast and transparent.

Integrating external ML services for advanced AI capabilities

Earlier in this chapter, we explored how Home Assistant can use historical data, rule-based logic, and simple machine learning techniques to learn habits, generate recommendations, and detect unusual behavior. In this section, we expand on those concepts by introducing additional examples that use external AI and machine learning services to provide more advanced reasoning, natural language understanding, and image analysis.

Rather than replacing Home Assistant's automation engine, these services complement it by analyzing summarized data, explaining complex situations, and generating structured recommendations for dashboards and automations. As discussed throughout this chapter, AI should act as an assistant rather than making decisions on its own. Language models can occasionally generate incorrect explanations or recommendations, so users should validate important decisions before converting them into automations.

Understanding AI integration options

Home Assistant provides several ways to integrate artificial intelligence, ranging from fully local solutions to cloud-based AI services. Some are included as native integrations, others are available through HACS, while advanced users can connect virtually any AI platform through REST APIs, Python scripts, Node-RED flows, or AppDaemon applications.

Choose the AI platform according to the task you want to accomplish. Some AI services are better at answering questions and having conversations, others specialize in analyzing images, while some are designed to run locally to provide privacy. *Table 12.5* presents AI options to be used in Home Assistant.

Option	Typical Use	Location
OpenAI, Anthropic Claude, Google Gemini	Natural language, summaries, automation explanations, Reasoning, structured reports, AI Tasks, Multimodal AI, image analysis, speech	Cloud
Ollama	Local conversational AI	Local
AI Task	Structured outputs from automations	Depends on LLM provider
MCP	AI tool integration framework	Hybrid
LLM Vision (HACS)	Camera and Frigate analysis	Local or Cloud
Node-RED/AppDaemon + APIs	Custom AI workflows	Hybrid

Table 12.5 – AI options for Home Assistant

We explored OpenAI and AI Task before. Claude and Gemini can be installed using direct Integrations as well as Ollama. The following subsections will detail some of the others presented in *Table 12.5*.

Note

Cloud AI providers typically charge per request or token, so running AI analyses only periodically or when significant events occur helps reduce your costs.

Using MCP to connect AI assistants

As AI assistants become more capable, they need a standard way to access external applications and tools. To address this, Anthropic introduced the Model Context Protocol (MCP) in November 2024 as an open standard that enables AI models to communicate with external systems through a common interface.

Instead of creating custom integrations for every AI provider, MCP standardizes how context, tools, and requests are exchanged. This makes it easier to connect multiple AI assistants while reducing integration complexity.

By using MCP to connect to Home Assistant, depending on the integration configured, an AI assistant can inspect entity states, analyze historical trends, create automations, retrieve dashboard information, or execute approved services.

Figure 12.12 shows how MCP can be integrated with Home Assistant.

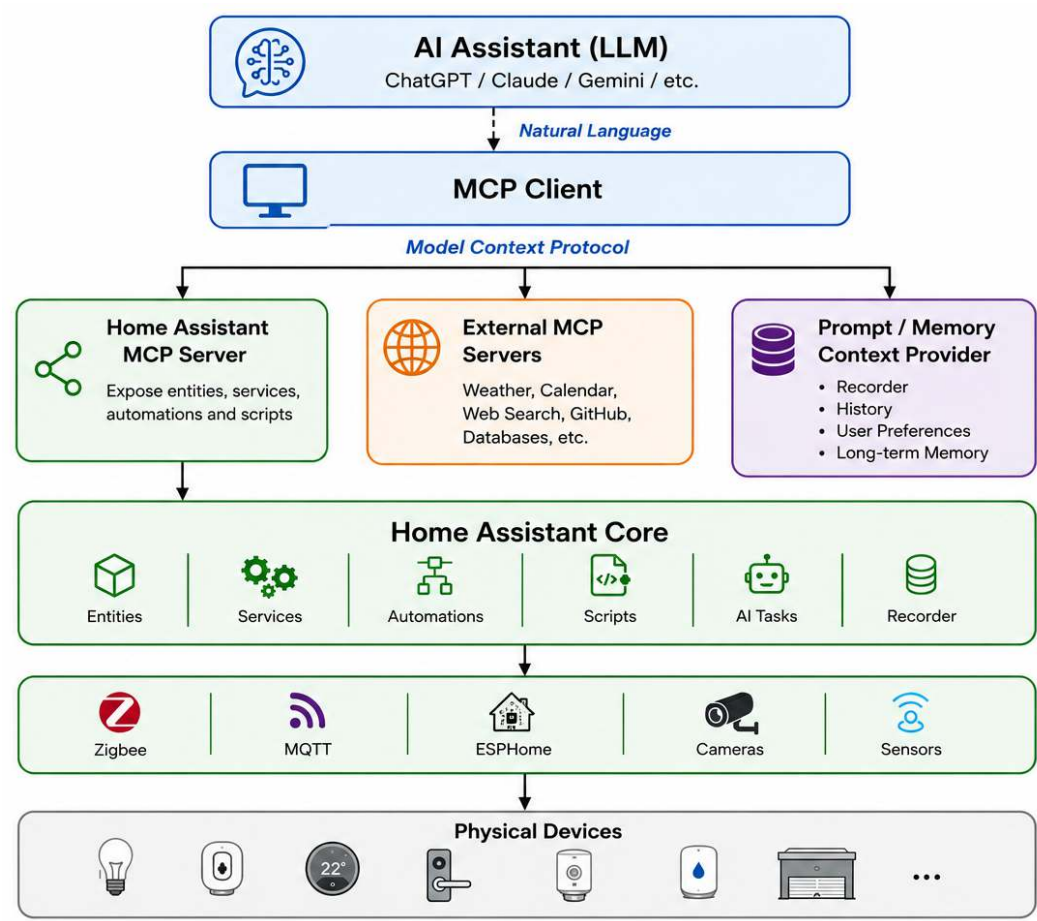


Figure 12.11 – MCP architecture for Home Assistant

How MCP works with Home Assistant, following *Figure 12.11*:

1. The user submits a request to an AI assistant.
2. The AI assistant sends the request through an MCP Client using the Model Context Protocol (MCP).
3. The MCP Client queries one or more MCP servers, depending on the information or tools required.
4. Data and context are returned from Home Assistant back to the AI assistant.
5. The AI assistant combines all available information to generate a response, explanation, recommendation, or proposed action.
6. If the request involves controlling devices, the AI assistant sends the approved action back through the Home Assistant MCP Server.
7. Home Assistant executes the requested service or automation.

Home Assistant includes a native MCP Server integration (https://www.home-assistant.io/integrations/mcp_server), however, at the time of writing, the native integration provides a limited set of features and configuration options. For the example in this chapter, HA-MCP (<https://github.com/homeassistant-ai/ha-mcp>), a community-developed MCP server that offers greater flexibility, supports more Home Assistant capabilities, and provides better control over which entities, services, and tools exposed to AI assistants will be used.

To configure and install HA-MCP in Home Assistant, follow the steps described here: <https://homeassistant-ai.github.io/ha-mcp/setup/>. There are instructions for different AI assistants and operating systems. I configured the HA-MCP as a custom component using AI client Claude Desktop and Local access. After the installation, if you go to Claude Desktop and ask if it can see your Home Assistant installation, you will have a summary of your installation like I had from mine on *Figure 12.12*.

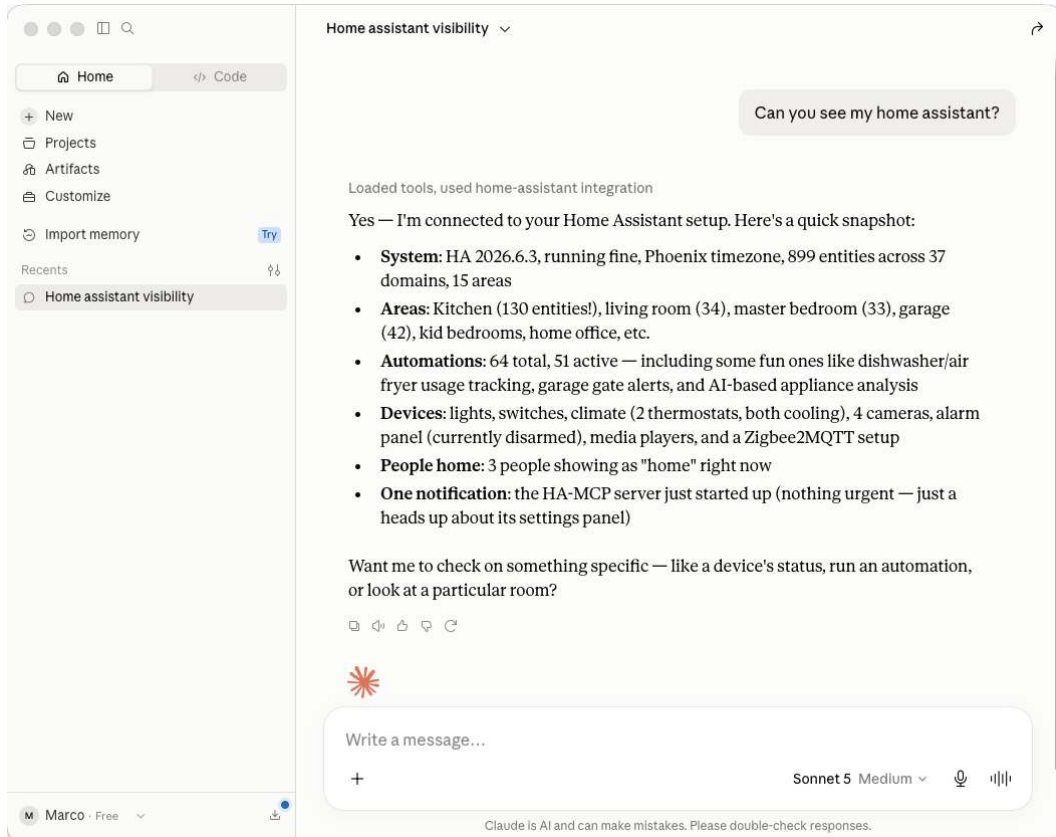


Figure 12.12 – Claude AI desktop assistant answer after configured with HA-MCP

After that, you can use your creativity to ask your AI Assistant questions about your Home Assistant installation. As an example, I asked about the issue with the Garage Refrigerator mentioned in the previous section, and, comparing it to other sensors in the garage, it detected an anomaly on July 12th and pointed to a "brief power blip or Wi-Fi/router hiccup in the garage". This assessment can be verified in the red rectangle presented in Figure 12.13.

Investigate the refrigerator_Garage energy today sensor entity for the last 30 days. Is there any anomaly detected? If yes try to correlate with other sensors installed in the garage and also the temperature in the backyard.

Synthesized energy consumption patterns and correlated temperature fluctuations

Here's what a 30-day check of **sensor.refrigerator_garage_energy_today** turned up (this is a Tasmota-style smart plug reporting daily kWh, resetting at midnight):

Overall pattern

- Daily consumption sat in a tight band all month — mostly **4.65–4.76 kWh/day** — with a gentle upward drift as the month progressed (early July ~4.65 kWh/day → late July ~4.73–4.76 kWh/day, roughly a 2–3% rise).
- Compressor duty-cycling looks mechanically normal: idle draw ~155–165 W, run spikes to ~330–520 W, cycling on/off every 10–15 minutes — no abnormal current or power spikes in the raw data.

The one real anomaly

- **2026-07-12** stands out: daily total dropped to **4.49 kWh** (well below the surrounding ~4.68–4.69 kWh days), and the underlying "state" value that day was oddly low (1.315) before resetting.
- This lines up exactly with device restarts logged that day: **GarageLights** (reason: Power On) and **GarageMotor** (reason: External System) both restarted at 2026-07-13T00:17 UTC (≈07-12 evening, local time). That points to a brief power blip or Wi-Fi/router hiccup in the garage — the refrigerator's plug likely dropped offline for part of the day, so the "today" counter under-reported rather than the fridge actually using less power. It's a data-continuity artifact, not an appliance issue.
- Today's entry (partial day, 2.946 kWh so far) looks "low" too, but that's just because the day isn't over yet — not a real anomaly. 

Figure 12.13 – Sensor anomaly detection and explanation using MCP Claude AI desktop

The Model Context Protocol enables AI assistants to interact with Home Assistant and external tools. This capability can complement anomaly detection by providing additional context that helps explain why an unusual event occurred.

Another rapidly growing application of AI in Home Assistant is camera analysis, where AI models can interpret images and generate meaningful descriptions of detected events.

AI-powered camera analysis

Camera-based AI is one of the fastest-growing areas of Home Assistant development. Instead of generating simple motion alerts, AI models can analyze snapshots or video frames and provide meaningful descriptions of detected events.

The main applications that use AI with cameras and integrate with Home Assistant are Frigate and LLM Vision.

Frigate

Frigate (<https://frigate.video/>) is an open-source Network Video Recorder (NVR) designed for local AI-based object detection. Instead of relying only on motion detection, Frigate uses artificial intelligence to recognize objects such as people, vehicles, bicycles, and packages directly on your local hardware. Since all video processing happens within your home network, camera footage does not need to be sent to cloud services, improving privacy.

Compared to a traditional NVR, Frigate provides more meaningful notifications because it can identify what actually triggered the event rather than simply detecting movement. It can also work alongside an existing NVR, adding AI capabilities without replacing your current system. Frigate can be installed either as a Home Assistant app or as a Docker container on a NAS, mini PC, or other supported hardware.

To integrate Frigate with Home Assistant, simply configure your cameras using their RTSP streams and install the Frigate integration through HACS. The integration communicates with Home Assistant using MQTT, making camera entities, object detection sensors, and zone information available for automations. Community Blueprints further simplify the creation of intelligent notifications, allowing alerts to be triggered only when specific objects, such as a person or vehicle, are detected in selected areas.

LLM Vision

LLM Vision (<https://llmvision.org/>) is a HACS integration that brings AI-powered image analysis to Home Assistant. Unlike Frigate, which continuously detects and tracks objects, LLM Vision is triggered by automations or events to analyze images, video clips, or camera snapshots and generate human-readable descriptions. It can also extract useful information, such as reading text or identifying specific objects, and update Home Assistant entities with the results.

LLM Vision works particularly well with Frigate. For example, when Frigate detects a person or vehicle, an automation can send the event to LLM Vision for a more detailed description before creating a notification. The integration is installed through HACS, configured with one or more AI providers such as OpenAI or Ollama, and can optionally use the LLM Vision Timeline Card to display AI-generated event summaries on a Home Assistant dashboard.

The examples presented in this section demonstrate how external AI services can extend Home Assistant beyond traditional automations. By combining local processing, cloud AI models, camera analysis, and standardized protocols such as MCP, it is possible to build a smart home that not only reacts to events but also understands context and provides meaningful recommendations. As these technologies continue to evolve, AI will become an increasingly valuable assistant for improving the comfort, efficiency, and security of smart homes and the future of Home Automation.

Summary

In this final chapter, you explored how artificial intelligence can extend Home Assistant beyond traditional rule-based automation. Rather than replacing deterministic automations, AI was presented as an assistant that learns from historical data, detects patterns, explains events, and recommends improvements.

The chapter began by introducing the differences between rule-based automation, statistical analysis, machine learning, and large language models, explaining where each technology fits within a smart home. It also discussed edge AI, cloud AI, and hybrid architectures, emphasizing that critical control should remain local while AI services provide higher-level reasoning and recommendations.

Building on these concepts, you learned how to transform Home Assistant historical data into AI-ready information by creating appliance sessions, generating summaries, and using Home Assistant AI Tasks to identify usage habits. The chapter demonstrated how AI-generated results can update helpers, create shadow entities, and produce practical recommendations through automations and notifications. It then expanded these ideas with rule-based recommendations, a simple machine learning implementation using K-Nearest Neighbors, anomaly detection techniques, and AI-assisted explanations that combine multiple sensors to provide meaningful context.

Finally, the chapter explored modern AI integrations, including OpenAI, Claude, AI Tasks, MCP, Frigate, and LLM Vision, showing how Home Assistant can interact with advanced AI services while maintaining privacy, security, and user approval. By the end of the chapter, you have a complete framework for building smart homes that not only automate tasks but also learn from behavior, detect unusual situations, and provide useful recommendations.

Final message

Throughout this book, we have gone far beyond simply connecting smart devices. You learned how Home Assistant works internally, optimized its performance, secured your network, built custom integrations, developed ESPHome devices, connected cloud platforms, designed advanced automation workflows, and finally incorporated artificial intelligence

The most important lesson is that a truly smart home is not defined by the number of devices it contains, but by how well those devices work together to improve everyday life. Technology should simplify your routines, increase efficiency, enhance security, and remain under your control.

Home Assistant and its community continue to evolve at an incredible pace. New hardware, integrations, AI capabilities, and communication protocols will continue to emerge, but the principles presented throughout this book will remain valuable for years to come.

I encourage you to continue experimenting, building your own devices, contributing to the Home Assistant community, and sharing your ideas with others. Every project you create will teach you something new, and every improvement will make your home a little smarter.

Thank you for joining me on this journey. I hope this book has inspired you not only to automate your home, but also to understand the technology behind it and confidently build the next generation of home automation solutions.

13

Unlock the Code Bundle and the PDF Version

Your copy of this book includes the following exclusive benefits:



Complete Code Bundle

Download the full source code for this book's examples and projects.



DRM-Free PDF Version

Download DRM-free PDF and ePub copies of this book.



7-Day Packt Library Access

Get 7-day unlimited access to 8,000+ books and videos. No credit card required.

Available for first-time Packt+ trial users only.



Next-Gen Reader Access

Read this book on Packt Reader with progress sync, dark mode and note-taking.

Follow this guide to unlock them. The process takes only a few minutes and only needs to be completed once.

Unlock this book's free benefits in three easy steps

Step 1

Have your purchase invoice ready for *step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit <https://www.packtpub.com/unlock-benefits/help>.

Note

Note: If you bought this book directly from the Packt website, no invoice is required. After *step 2*, you can access your exclusive content right away.

Step 2

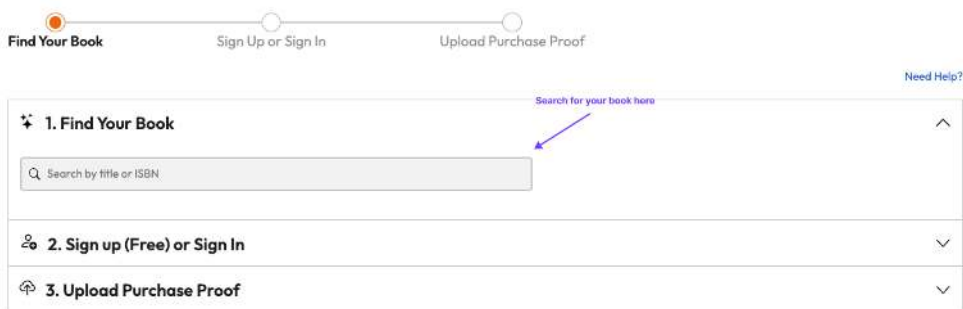
Scan the QR code or go to packtpub.com/unlock.



On the page that opens (similar to *Figure 13.1* on desktop), search for this book by name and select the correct edition.

Unlock Your Book's Free Benefits

Bought a Packt book from Amazon or one of our channel partners? Unlock your free benefits in 3 easy steps.



The image shows the Packt unlock landing page on a desktop. At the top, there's a progress bar with three steps: 'Find Your Book' (active, orange circle), 'Sign Up or Sign In' (grey circle), and 'Upload Purchase Proof' (grey circle). Below the progress bar, there's a search bar with the placeholder text 'Search by title or ISBN'. A purple arrow points to the search bar with the text 'Search for your book here'. To the right of the search bar, there's a link that says 'Need Help?'. Below the search bar, there are three steps listed: '1. Find Your Book' (with a star icon), '2. Sign up (Free) or Sign In' (with a person icon), and '3. Upload Purchase Proof' (with a document icon). Each step has a dropdown arrow on the right.

Figure 13.1: Packt unlock landing page on desktop

Step 3

After selecting the book, sign in to your Packt account or create one for free. Then, upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need help?

If you get stuck and need help, visit <https://www.packtpub.com/unlock-benefits/help> for a detailed FAQ on how to find your invoice and more. This QR code will take you to the help page.



Note

Note: If you are still facing issues, reach out to customercare@packt.com.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

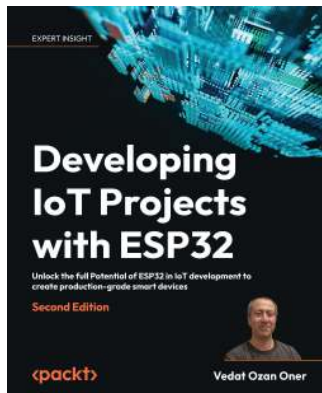
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Developing IoT Projects with ESP32

Vedat Ozan Oner

ISBN: 978-1-80323-768-8

- Explore ESP32 with IDE and debugging tools for effective IoT creation
- Drive GPIO, I2C, multimedia, and storage for seamless integration of external devices
- Utilize handy IoT libraries to enhance your ESP32 projects
- Manage WiFi like a pro with STA AP modes, provisioning, and ESP Rainmaker framework features
- Ensure robust IoT security with secure boot and OTA firmware updates
- Harness AWS IoT for data handling and achieve stunning visualization using Grafana
- Enhance your projects with voice capabilities using ESP AFE and Speech Recognition
- Innovate with tinyML on ESP32-S3 and the Edge Impulse platform



Building Smart Home Automation Solutions with Home Assistant

Marco Carvalho

ISBN: 978-1-80181-529-1

- Understand the fundamental concepts of home automation systems
- Set up a home automation system using Home Assistant and Raspberry Pi
- Create and configure ESP8266-based sensors to work with Home Assistant
- Hack a commercial actuator to work with Home Assistant using Tasmota
- Create automations, customize, and use applications with Home Assistant
- Leverage IoT software tools to take your home automation to the next level
- Work on hands-on projects, including LED strip lights and an ESP32 five-zone temperature logger
- Explore home automation FAQs, emerging technologies, and trends

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packt.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now that you've finished *Advanced Smart Home Automation Solutions with Home Assistant*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



<https://packt.link/r/1806384698>

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

AI and ML, in home automation	346, 347
edge AI, versus cloud AI	349, 350
limitations	351
LLMs	348
machine learning	348
practical possibilities	352
rules	348
statistical analysis	348
AI-driven automation	
external ML services, integrating	373
habit learning, from Home Assistant data	352
rule-based and simple ML-driven suggestions	365
AI-powered camera analysis	379
Frigate	379
LLM Vision	379
AWS IoT Core	171
URL	171
Alexa Media Player	335
AppDaemon	296 - 298
application structure	299 - 301
HADashboard	303
installing	298, 299
reusability	301, 302
Arduino Cloud	172
URL	172
advanced templating	72
conditionals	72
loops	75 - 77
aliases	33
anchors	33
attributes	68
automation design patterns	289

layered automation	289, 290
architecture	
multi-step automations	293, 294
sequences	293, 294
state logic, separating from action logic	292, 293
state machines and conditional flows, combining	294 - 296
state machines, in home automation	290 - 292
automation tool comparison	
features	309, 310
hybrid architecture	311, 312
practical example	312
selection decision guidelines	310, 311

B

Battery Notes	334
Bluetooth/BLE	125
Buildroot	
URL	7
backup files, retrieving with GitHub	47
dashboard integration	54
helper development	50
script development	51 - 53
shell command	48, 49
shell script development	48, 49
backup routine	
creating, with GitHub	45, 46
built-in debugging tools	
using	58

C

Catppuccin	337
Cloudflare Tunnel	113
camera-based AI	379
car GPS virtual device	

creating, with MQTT discovery	229 – 231	device renaming	116
cloud AI	350	devices connecting to network	
advantages	155	monitoring	116
cloud IoT platforms	154	direct port forwarding	
Arduino Cloud	172	risks	111
AWS IoT Core	171	dual measurement sensor integration	
comparison	172 – 174	creating, for HACS	339, 340
Microsoft Azure IoT Hub	171	dynamic data extraction	67
Particle Cloud	172	dynamic data manipulation	67
use cases	155 – 157		
conditionals	72	E	
indirect implementation	72 – 74	ESP32 camera sensors	271 – 273
nested if/elif/else	74	ESP32 device	
configuration		connecting, to Golioth cloud	162 – 165
modularizing	40 – 43	ESP32-S3 camera	277
core Golioth components	157	hardware architecture	277, 278
APIs	159	HTTP-based camera	278 – 280
device identity	158	interface feature	
Embedded devices	159	integrating, with Home Assistant	281, 282
OTA firmware updates	159	network independence,	280
secure communication	158	with mDNS	
settings service	159	software features	278
telemetry streams	159	Wi-Fi provisioning, via captive portal	280, 281
web console	159	ESP32-S3-Touch-LCD-3.5 board	253
curated advanced HACS content	331	ESP32-based temperature sensor circuit	160
advanced dashboard cards	332, 333	flowchart	161
custom device integrations	335	ESPHome	181
recommended categories, for advanced users	331, 332	Auto Code Generation	185
system monitoring	333, 334	Build and upload Firmware	186
integrations		communication buses	182 – 184
themes, for usability and wall-panel dashboards	337	comparing, with traditional firmware development	184, 185
		core concepts	185
D		Deploy and Monitor	186
Deep Sleep		device discovery	196 – 198
power consumption,	201 – 203	entity creation	196 – 198
optimizing		hardware platforms	182, 183
Dongle Lite MG21	133	in Home Assistant systems	181, 182
Double Measurement Sensor	11	lambdas	198
DuckDNS and Let's Encrypt			
SSL certificates, managing	113		
data flow	18, 19		
defense in depth	100, 101		

on-device automation,	199 – 201	configuration	236
implementing with actions		data	237
and triggers		data, integrating into Home	242 – 247
peripherals	182, 183	Assistant	
power consumption,	201 – 203	hardware architecture	235, 236
optimizing with Deep Sleep		sentences	237
templates	198	system architecture	236
URL	181	GitHub	
YAML Configuration	185	backup files, retrieving	47
ESPHome API		backup routine, creating	45, 46
URL	196	Golioth	157
ESPHome LVGL Designer	259 – 262	Golioth cloud	
tool		ESP32 device, connecting to	162 – 165
ESPHome device		Golioth data	
code, compiling	194	integrating, with Home	165
code, generating	190 – 193	Assistant	
code, uploading	194, 195	Google AI Edge LiteRT documentation	
configuring	187 – 189	reference link	350
Edge Impulse	350	groups	68
edge AI	349	H	
entity lists		HACS content	
selecting	69	configuring	328
transforming	69	installing	327
external ML services, for	373	troubleshooting	329, 330
advanced AI capabilities		HADashboard, in	303
AI integration options	373, 374	AppDaemon	
AI-powered camera	379	creating	303, 304
analysis		enabling	303, 304
MCP, for connecting AI	374 – 377	Home Assistant	62, 155
assistants		alerts, building with	118, 119
F		automations and	
Field of View (FOV)	272	notifications	
Figma	259	apps	8
FreeCAD		database maintenance	25 – 27
URL	143	database optimization	21 – 23
Frigate	379	ESPHome	181, 182
filters, Jinja2	64 – 66	event monitoring	117
firmware updates	114	Golioth data, integrating	165
folder structure		with	
best practices	43, 44	incident response	119
G		workflow, creating	
GPS fix	237	integration minimalism	28
GPS tracking and	234	integrations	10
location-based		log monitoring	117
automations		logging configuration	27, 28
		noisy entities, identifying	23, 24
		noisy events, identifying	23, 24

resource monitoring	28, 29	cameras, integrating	271
secrets.yaml, using	55	display technologies	252
suspicious behavior, detecting	118	hardware interfacing	253
Home Assistant	324	system architecture	250, 251
Community Store (HACS)		touch technologies	252
AppDaemon apps	325	Home automation	
custom component	337, 338	real-world breaches	99
structure		threat actors	98, 99
dashboard cards	325	habit learning, from	352
development basics	337, 338	Home Assistant data	
dual measurement sensor	339, 340	AI Task examples	364
integration, creating		AI Task, using	357 – 359
ecosystem	340	data sources and quality	353, 354
installation	326	helpers, updating from AI results	360, 361
integrations	325	notifications, sending from	362, 363
main screen	326, 327	shadow sensors	
maintenance checklist	341, 342	periodic AI trend analysis	362, 363
templates	325	sensor data, preparing	355, 356
themes	325	shadow entities, creating	361, 362
update workflow	341	hybrid security	312
Home Assistant	5, 6	automation example	
Container		AppDaemon processing	316
Home Assistant Core	9	flow layer	314, 315
components	9	Home Assistant helpers and groups	313
Event Bus	9, 20	logic layer	316
Service Registry	9, 20	Node-RED flow	314, 315
Set state	20	state layer	313
State Machine	9, 20	testing	317
Timer	9		
Home Assistant	5 - 7	I	
Operating System (HAOS)		IoT and Smart Home Systems	
Home Assistant	8	vulnerabilities	97, 98
Supervisor		iOS Themes	337
Home Assistant Template Editor	78, 79	include directives	34 - 39
errors and fixes	79 – 81	benefits	34
Home Assistant automations	288	integrations, Home Assistant	
basic, versus advanced	289	action	20
automation		action and event-driven	10
Home Assistant configuration		integrations	
managing	44	automation	19
Home Assistant frontend	18	Binary Sensor	19
Home Assistant project		computed entity	10
camera sensor	273	integrations	
camera sensor module, configuring	273 – 276	custom integration	12 – 17
		development	

-
- custom integration, 17, 18
 - converting to core
 - integration
 - developing 11
 - external device and service 10
 - integrations
 - Internet of Things (IoT) 10
 - device integrations
 - Tasmota Integration 20
 - trigger 20
 - Zigbee Integration 19
- ## J
- Jinja2** **62, 63**
 - filters 64 – 66
 - syntax examples 63
 - templates 62, 63
 - tests 66
 - URL 64
 - variables 64
- ## L
- LLM Vision** **379, 380**
 - LQI (Link Quality Indicator)** **139**
 - LVGL (Light and Versatile Graphics Library)** **259**
 - LibreTiny project**
 - URL 183
 - LocalTuya** **336**
 - loops**
 - for ... in 75 – 77
 - Itchiptool**
 - URL 189
- ## M
- MQTT Explorer** **221 - 223**
 - message metadata 224 – 226
 - payloads, inspecting 224 – 226
 - test messages, publishing 226, 227
 - URL 222
 - MQTT GPS solution sensor software implementation** **238**
 - connectivity management 241
 - core software structure 239
 - diagnostics data 239
 - GPS parsing functions 240
 - key function overview 240
 - LED feedback system 241, 242
 - MQTT publishing functions 241
 - MQTT topic design 238
 - raw GPS data 238
 - Status Topic 238
 - system flow 239
 - UART initialization 241
 - MQTT discovery**
 - car GPS virtual device, creating 229 – 231
 - MQTTX** **227**
 - overview 228, 229
 - URL 228
 - MQTTX, for testing GPS** **232**
 - MQTT topics and automations**
 - automation testing 233, 234
 - connectivity test 232
 - garage approach detection 233
 - test
 - GPS data validation 233
 - Message Queuing Telemetry Transport (MQTT)** **214**
 - communication model 215
 - components 215
 - Quality of Service (QoS) 217
 - topic hierarchies 216
 - topics 216
 - wildcard characters 217
 - Microsoft Azure IoT Hub** **171**
 - URL 171
 - Mosquitto** **218**
 - authentication 220
 - basic configuration 219
 - installation 219
 - security 220
 - Mushroom Themes** **337**
- ## N
- Nabu Casa** **113**
 - URL 113
 - Noctis** **337**
 - Node-RED** **305, 306**
 - installing 306, 307
 - node examples 308

node groups	308
URL	308
using	306, 307

naming conventions	
best practices	43, 44

native ESPHome API	
using	195

nested if/elif/else	74
----------------------------	-----------

numeric calculations	70, 71
-----------------------------	---------------

O

OpenVPN	111
----------------	------------

OpenWRT	
URL	103

P

Particle Cloud	172
URL	172

passwords	
handling	115

R

RSSI	139
-------------	------------

refrigerator garage energy monitoring solution	81
dashboard	88 – 90
energy price calculation	83, 84
refrigerator energy and cost	84 – 86
threshold and notification	86 – 88

resilient home automation systems	
designing	318 – 320

rule-based and simple ML-driven suggestions	
AI, as explanation layer	371, 372
anomaly detection best practices	373
anomaly detection, for security and efficiency	370
common smart home anomalies	370
implementing	365
rule-based suggestion	365 – 367
simple machine learning	368 – 370
recommendation, implementing	

S

SSL certificates	
managing, with DuckDNS and Let's Encrypt	113

Smart Casa Monitor device	251
----------------------------------	------------

Smart Trash Collection Monitor	
building, with MPU-6500	203
ESPHome firmware	205 – 210
hardware layer	204, 205
Home Assistant	210, 211
automation logic	

SonoffLAN	336
------------------	------------

Spook	334
--------------	------------

secrets management	55
---------------------------	-----------

secrets.yaml	
using, in Home Assistant	55

secure OTA updates	116
---------------------------	------------

segmented smart home network	
designing	102
firewall configuration	106 – 109
implementing	102
interfaces, creating	104, 105
IoT wireless interface, enabling	109, 110
router, selecting	103
traffic routes	106 – 109
VLANs, setting up	104, 105

sensitive data handling	55
--------------------------------	-----------

slow system	
diagnosing	29

state management	18, 19
-------------------------	---------------

states	68
---------------	-----------

T

TFT touchscreen software application	
building, with ESPHome and MQTT	254, 255
configuration, storing with global variables	267, 268
display, configuring in ESPHome	256 – 258
functional requirements	254

MQTT publish	266, 267	server mode	111
functionality,		Virtual Local Area Network (VLAN)	101
implementing		variables, Jinja2	64
MQTT, configuring	259	W	
navigation between main	263, 264	Watchman	334
and settings pages,		Wi-Fi	125
implementing		WireGuard	111
navigation, implementing	262	wireless protocols, home automation	
prototype mockup	255	comparing	124
screen layouts, building	261, 262	X	
with ESPHome LVGL Designer		Xiaomi Miot	336
shared text editor	264 – 266	Y	
status indicator icons,	269	YAML	32
creating for clock		debugging	59
status indicator icons,	268	errors	57, 58
creating for MQTT		files	33
status indicator icons,	268	parsing	56, 57
creating for Wi-Fi		syntax	33
timer feature,	269 – 271	troubleshooting	56
implementing		Z	
touchscreen, configuring in	256 – 258	Z-Wave	125, 140
ESPHome		drawbacks	141
user interface, building	259, 260	Zigbee	124, 125
with LVGL support		communication channels	128, 129
Thread	125	coordinator	126
TinyML	349	coordinator placement best	137
URL	349	practices	
temperature sensor		device types	125
configuring, to output	166	end device	127
Goloth states		mesh improvements,	139, 140
data, pulling from Goloth	169, 170	evaluating	
cloud to Home Assistant		mesh network	127, 128
OTA feature, adding	167, 168	poor-quality devices,	139, 140
templating, Jinja2	62, 63	identifying	
tests, Jinja2	66	router	126
threat actors	98	router selection	137
U		Zigbee Home Automation (ZHA)	130, 133
UI Lovelace Minimalist	337	device pairing	134
unused services		installation	133, 134
disabling	115	optimization	135
V			
VPN-based remote access	111		
client mode	112		

versus Zigbee2MQTT (Z2M)	135, 136	Zigbee signals	
		diagnosing, with LQI and RSSI	138, 139
Zigbee Motion Detector	19		
Zigbee for garage gate control	141 - 143, 147 - 150	Zigbee2MQTT (Z2M)	130
electronics board	146, 147	binding	132
fast prototyping	143, 146	firmware flashing	130, 131
mechanics	143, 146	groups	133
Zigbee network		installation	130, 131
right number of routers, determining	137, 138	link quality metrics	133
		network configuration	131
		Over-the-Air (OTA)	133
		pairing devices	131

