SMART HOME EVENT NOTIFICATION SYSTEM USING MESSAGE QUEUES AND REST APIS

PROFESSOR LETTERA GALLETTA

SCHOOL OF MATHEMATICAL, PHYSICAL AND NATURAL SCIENCES
SOFTWARE: SCIENCE AND TECHNOLOGY
B032427 (B255) - DISTRIBUTED PROGRAMMING FOR WEB, IOT AND MOBILE SYSTEMS 2024-2025

Syed Mohammad Askari Abidi
Software Engineer – Matriculation: 7178458
Email: syed.abidi@edu.unifi.it

UNIVERSITÀ DEGLI STUDI FIRENZE

Table of Contents

Project Overview

The Smart Home Event Notification System using Message Queues and REST APIs is a distributed software application developed in Go. It simulates a smart home environment where multiple sensors (such as motion detectors and temperature monitors) continuously generate events that are transmitted asynchronously through a message queue (RabbitMQ).

The primary objective of this project is to demonstrate real-time communication and event-driven architecture within a distributed system. Incoming sensor events are processed by a Go-based backend service, persisted into an SQLite database, and made accessible via RESTful APIs and a clean web dashboard.

System Architecture

The architecture of the Smart Home Event Notification System follows a modular, event-driven, and loosely coupled design, combining message queues with REST APIs and a lightweight frontend.

The system is composed of the following key components:

Sensor Simulator

1. Mimics real-world sensors like motion detectors and temperature readers.
2. Sends events (e.g., "motion detected", "27.5°C") to $RabbitMQ$ using a $POST$ request via the $/simulate$ endpoint.

RabbitMQ Message Broker

1. Acts as the central messaging backbone.
2. Decouples the sensor event producers from the event consumers.
3. Ensures asynchronous and reliable delivery of sensor data.

Go-based Consumer Service

1. Listens to the RabbitMQ queue for incoming sensor events.
2. Parses the events and persists them in a local SQLite database.
3. Runs background goroutines for real-time processing.

SQLite Database

1. Stores sensor data with fields like sensor type, value, and timestamp.
2. Enables querying for logs, latest values, and system status.

RESTful API Layer

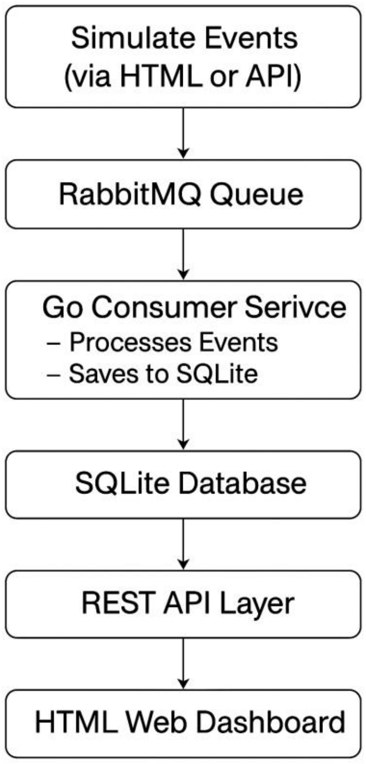
Provides HTTP endpoints for:

1. $/logs$ – full event history
2. $/status$ – latest value per sensor
3. $/simulate$ – event triggering interface
4. $/dashboard$ – current sensor view

Web Dashboard (HTML + Go Templates)

1. Renders the system state in a clean UI.
2. Displays real-time sensor values with auto-refresh.
3. Allows users to simulate events interactively.

Architectural Diagram



Technologies and Tools

The project utilizes a modern, lightweight, and modular technology stack designed to simulate a real-world distributed system with asynchronous messaging, RESTful APIs, and frontend integration. Below is a breakdown of the tools and technologies used:

- Programming Language
  - Go (Golang)
    - Chosen for its performance, concurrency support, and suitability for building scalable backend systems.
- Messaging System
  - RabbitMQ
    - A robust message broker that handles asynchronous communication between the sensor simulator and the backend service. It decouples producers from consumers and ensures message delivery.
- Database
  - SQLite
    - A lightweight, file-based relational database used to persist incoming sensor events. Suitable for local development and fast prototyping.
- Web Framework
  - Go net/http and html/template
    - Used to build RESTful APIs and render dynamic HTML templates for the web dashboard and simulate interface.
- Containerization
  - Docker & Docker Compose
    - Used to containerize the entire system (Go app + RabbitMQ), making it portable and easy to deploy. Docker Compose ensures the services are orchestrated seamlessly.
- Development Tools
  - Visual Studio Code
    - As the primary IDE for writing and organizing Go code and templates.
- Git & GitHub
  - For version control and source code management.

These tools enabled the development of a modular, event-driven, and easily deployable distributed system suitable for smart home applications.

Topics Covered

This project incorporates several key concepts and techniques taught in the Distributed Programming for Web, IoT, and Mobile Systems course. Below is a mapping of course topics to implemented features:

Course Topic	**How It Was Covered in the Project**
Message Queues & Event-Driven Design	Implemented using $RabbitMQ$ to decouple sensor producers from consumers.
RESTful Web Services	APIs for accessing logs, system status, and triggering sensor events.
Concurrency in Distributed Systems	Used Go routines to process messages and serve HTTP requests concurrently.
Service Modularity	Project structured using cmd, $internal/api$, $internal/sensor$, etc.
Lightweight Web Interfaces	Built with Go html/template to render dashboard and simulate pages.
Containerization with Docker	Entire system runs via Docker Compose including message broker and backend.
Data Persistence in Distributed Apps	Used SQLite to store and retrieve event logs and sensor statuses.

This practical implementation reflects a hands-on understanding of how to build, deploy, and interact with distributed applications using modern tools and patterns.

Use Case Description

Scenario: Smart Home Monitoring System

Imagine a small smart home setup where IoT devices (e.g., motion detectors and temperature sensors) are continuously monitoring the environment. These devices send data such as:

- "Motion detected in the living room"
- "Temperature reading: 27.5°C"

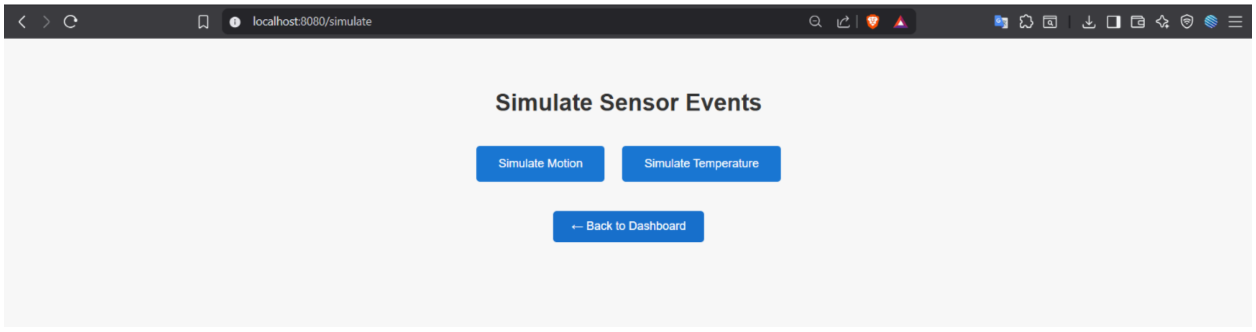
In this system:

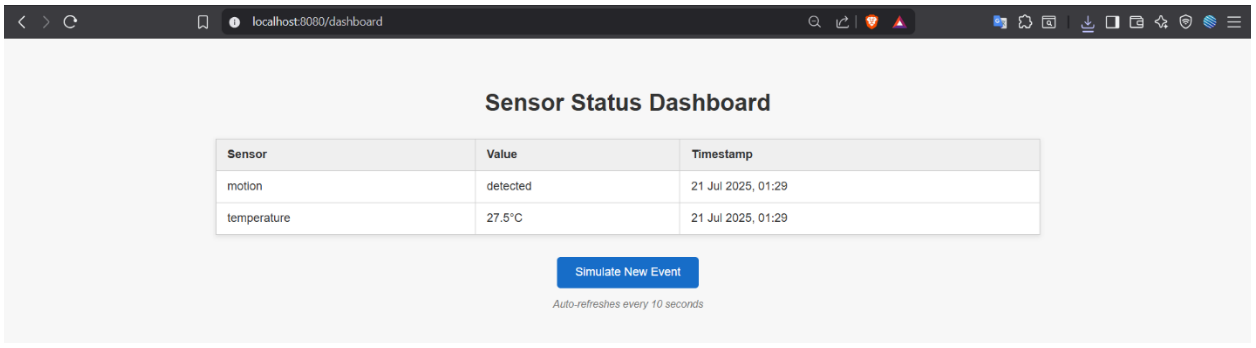
- Events are published asynchronously to a RabbitMQ queue.
- A Go consumer service picks up these events and stores them in a database.
- A dashboard UI allows users to:
  - Monitor the latest sensor status.
  - View historical logs of events.
  - Manually simulate new sensor events for testing.

This is particularly useful in:

- Smart homes
- Industrial IoT setups
- Environmental monitoring systems

The modular design makes it extensible to more sensor types or real IoT device integration in the future.

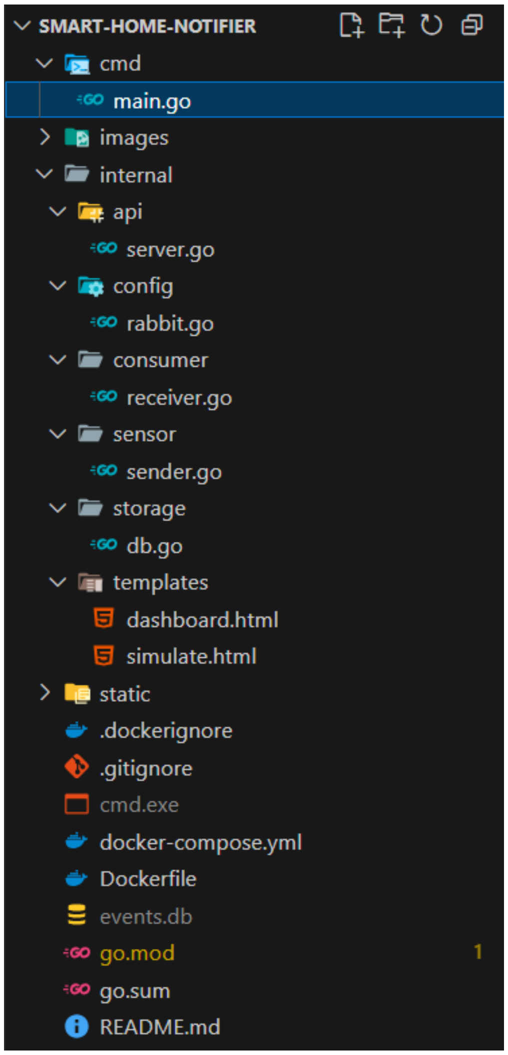




Sensor	Value	Timestamp
motion	detected	21 Jul 2025, 01:29
temperature	27.5°C	21 Jul 2025, 01:29

Design and Implementation

The project follows modular, scalable architecture, leveraging Go's idiomatic project structure. All major components are clearly separated by concern to enhance maintainability and readability.



```
SMART-HOME-NOTIFIER
  cmd
    main.go
  images
  internal
    api
      server.go
    config
      rabbit.go
    consumer
      receiver.go
    sensor
      sender.go
    storage
      db.go
    templates
      dashboard.html
      simulate.html
  static
  .dockerignore
  .gitignore
  cmd.exe
  docker-compose.yml
  Dockerfile
  events.db
  go.mod                1
  go.sum
  README.md
```

Event Flow Summary

1. Event Simulation:
    a. Users simulate motion or temperature events via $/simulate$ HTML page.
    b. Events are sent to the RabbitMQ queue using Go's $sender.go$.
2. Asynchronous Processing:
    a. $receiver.go$ runs a consumer that listens to the RabbitMQ queue.
    b. Incoming events are logged and stored into $events.db$ via $db.go$.
3. API and Web Access:
    a. $server.go$ registers HTTP routes (/, $/logs$, $/status$, $/simulate$, $/dashboard$).
    b. $dashboard.html$ displays current sensor status; simulate.html lets users trigger events.
    c. HTML rendered with Go's $html/template$.
4. Persistence:
    a. All sensor data is stored using SQLite ($events.db$) for logging and querying.
5. Deployment:
    a. $Dockerfile$ and $docker-compose.yml$ ensure the system (Go app + RabbitMQ) is containerized and portable.

Why This Design?

- Clean Separation of Concerns: Each package (api, sensor, consumer, etc.) handles one job.
- Loose Coupling: The producer (sensor) and consumer are decoupled using RabbitMQ.
- Extensible: More sensor types and routes can be added with minimal changes.
- Testable: The modular structure makes unit testing individual parts straightforward.
- Deployable: Docker makes deployment on any machine consistent and reproducible.

Instructions to Compile, Run, and Deploy

This project is fully containerized using Docker and can be run locally or deployed to any Docker-compatible environment.

PREREQUISITES

- Go version ≥ 1.24.5
- Docker Desktop
- Git (optional, for cloning from GitHub)

COMPILE AND RUN LOCALLY (OPTIONAL)

If you want to run it without Docker:

$go\ mod\ tidy$

$go\ run\ ./cmd$

Then visit:

- $http://localhost:8080$ → API Home
- $http://localhost:8080/dashboard$ → Dashboard
- $http://localhost:8080/simulate$ → Simulate Events

Ensure RabbitMQ is running on $amqp://guest:guest@localhost:5672/$.

RUN USING DOCKER

Make sure Docker is installed and running.

Clone the Project:

$git\ clone\ https://github.com/askariabidi/smart - home - notifier.git$

$cd\ smart - home - notifier$

Run the App with Docker Compose:

$docker - compose\ up - build$

This will:

- Start the Go backend
- Start a RabbitMQ server
- Bind everything on $localhost:8080$

Access the System:

- API Base: $http://localhost:8080$
- Dashboard: $http://localhost:8080/dashboard$

- Simulate Sensor Events: $http://localhost:8080/simulate$
- Logs (JSON): $http://localhost:8080/logs$
- Sensor Status (JSON): $http://localhost:8080/status$
- RabbitMQ Admin Panel: $http://localhost:15672$
    - (username: guest, password: guest)

Deployment Options

- GitHub Codespaces: Upload the repo and $run\ docker - compose\ up\ --build$ in terminal.
- Render.com / Railway / Heroku (Docker): Configure Docker deployment and environment.
- Raspberry Pi / IoT Gateway: Since Go binaries are cross-platform, you can compile and run on edge devices.

Conclusion

The Smart Home Event Notification System successfully demonstrates the key principles of distributed systems by integrating asynchronous communication, concurrency, and modular service design. The project simulates a real-world scenario where sensors generate data continuously, and a backend system processes and visualizes this data in real time.

Throughout this project, I applied several concepts learned during the Distributed Programming for Web, IoT, and Mobile Systems course under the guidance of Professor Letterio Galletta, including:

- Event-driven programming using RabbitMQ
- Service modularization using Go
- RESTful API development
- Frontend rendering using Go’s template engine
- Deployment through Docker and Docker Compose

This project not only strengthened my understanding of distributed system architecture but also equipped me with practical skills in building real-world, deployable applications. Its extensibility allows future integration with actual IoT sensors or cloud infrastructure, making it a valuable base for continued learning and experimentation.