Mpm-Smart Documentation

Our very smart smart-home app

Team-Mpm 2024

Table of Contents

Home

The Team

- Plazinski Olaf (Team Lead)
- Martin Gabriel (Tech)
- Mayer Julian (UI)
- Toifl Jacob (UI)

The idea

Creating a fully wireless smart-home/IOT automation system with a central management controller. We implement an open standard built on plugins and an extensible architecture. The entire Project is fully open source, and contributions from the community are highly welcome. We collect metrics and telemetry from multiple sources and display them in custom configurable dashboards. We also plan to support actor devices that can be controlled via the dashboard in real time or automatically controlled via dynamic workflows.

User Guide

Initial Setup

User Management

Devices Integration

Plugin Management

Developer Documentation

This document is intended for developers who want to contribute to the Writerside project. It provides an overview of the project's architecture, the technologies used, and the development process.

Required Tools

- .NET 9 SDK
- Dotnet ef tools (For Entity Framework Core)
- Platform IO (For ESP32 development)
- JetBrains Writerside (Documentation)
- Docker
- A modern web browser
- Git
- Make

Optional Tools

- Azd
- Rider
- Visual Studio Code
- cppcheck

GitHub Actions

Automatic Testing

The project uses GitHub Actions to run tests on every push/pr to the master branch. These tests are mandatory and must pass before a PR can be merged. When contributing new code, also try to achieve full test coverage where possible.

Continuous Integration

The server image will be automatically built and published on push to the master branch. The documentation will be published on GitHub pages.

In the future, there will be also plans to automatically publish all the base packages on nuget and build debian packages, including a full raspberry pi image.

Code Quality

The project uses GitHub Actions to run code quality checks on every push/pr to the master branch. These checks are mandatory and must pass before a PR can be merged.

Getting the Code

To get the code, clone the repository:

```
git clone --recursive https://github.com/Team-MPM/MPM-Smart
```

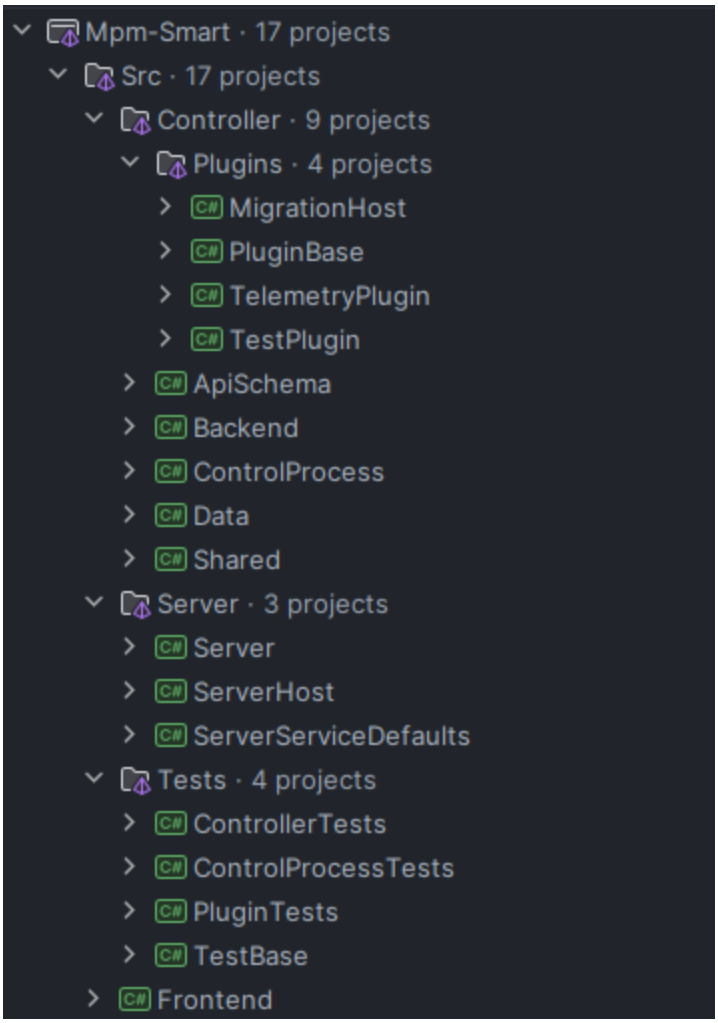
Architecture

Introduction

The solution is generally organized into the following parts:

- Dotnet Solution **.sln/.slnx**
- Visual Studio Code Workspace **.code-workspace** for Platform IO

Dotnet Solution Architecture



solution_architecture

Frontend

The frontend is responsible for rendering the user interface and handling user input.

The frontend is a Blazor WebAssembly application. It is written in C# and runs in the browser. It makes use of the PWA (Progressive Web App) features to provide an offline-first experience.

We primarily make use of the following libraries:

- MudBlazor
- FluentValidation

Backend

The backend is the primary application component running on the controller unit. It is responsible for loading all the plugins and managing the communication between them. It handles the communication between devices and acts as a bridge between the frontend and the plugins.

Control Process

Manages the lifetime of the backend process and handles plugin updates and automatic restarts.

Api Schema

Shared class library that contains the data transfer objects (DTOs) used by the frontend and the backend.

Shared

Shared class library that contains the shared code between the backend and the plugin layer.

Plugin Base

Base code and Interfaces for Plugins to reference.

Data

Data Layer of the core Backend Application containing all EF Core Entities and the Database Context configurations.

Server/ServerHost

THe Plugin Server providing Plugins for download via Azure Cloud Storage. It also contains the documentation reference and a List of all Mpm-Smart devices.

Tests

Solution folder containing all the Unit tests separated into the different layers. There is also a **TestBase** class library that provides a unified testing experience. We primarily use XUnit for testing, FluentAssertion for assertions, and Moq for mocking.

Build Instructions

To build the dotnet solution, open the solution file in the IDE of your choice or build it from the command line using the `dotnet build` command. Node: Even when running only a specific project, performing a full dotnet build on the solution first is recommended so that all the plugins and dependencies are built correctly the first time you run them locally.

To build the device firmware, open the Platform IO workspace in Visual Studio Code and build the project using the Platform IO build command. Alternatively, you can, of course, use Platform IO on the command line as well.

To build the documentation as PDF, open the JetBrains Writerside workspace and build the documentation using the included run configuration.

The Makefile in the root of the repository provides shortcuts for mose these commands.

To build the debian packages and controller images from source, consult the `make help` command. Note that this requires you to be on a linux system and have the specified dependencies installed.

To run the frontend in release mode locally, perform a `dotnet publish -c Release` on the frontend, then navigate to the output directory based on `bin/Release/net9.0/publish/wwwroot` and run a local webserver to serve the files in this directory (tested using apache2 and nginx). This can easily be achieved using docker: `docker run -it --rm -p 8080:80 --name mpm-smart-app -v .:/usr/local/apache2/htdocs:ro httpd`

Plugins

Overview

Plugins are a way to extend the functionality of the Core Application Layer provided by the Mpm-Smart ecosystem. Plugins are usually provided by a Plugin Server or loaded from a local directory on the controller. Plugins are exclusive to the backend layer. In the UI, all plugins can implement a set of options and views providing user interactivity for plugin-specific settings and exposed data.

How plugins get loaded

Plugins are loaded by the Plugin Manager. The Plugin Manager is responsible for loading, unloading, and managing the lifecycle of plugins. All Plugins are loaded at startup and are initialized in no specific order. Currently, there is no way of reloading plugins while the backend service is running. In deployment, this update workflow is automated by the control process shipping with the backend.

Creating your first Plugin

Introduction

Create an empty C# dotnet 9.0 class library and adapt the **Plugin.csproj** file to the following:

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFramework>net9.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
    </PropertyGroup>

    <PropertyGroup>
        <OutputPath>your/plugin/output/path</OutputPath>
        <AppendTargetFrameworkToOutputPath>false</AppendTargetFrameworkToOut
putPath>
    </PropertyGroup>

    <ItemGroup>
        <ProjectReference Include="..\plugin_base\PluginBase.csproj" Private
="False"/>
        <ProjectReference Include="..\..\Shared\Shared.csproj" Private="Fals
e"/>
    </ItemGroup>

    <ItemGroup>
        <FrameworkReference Include="Microsoft.AspNetCore.App"/>
    </ItemGroup>

    <ItemGroup>
        <None Update="plugin.json">
            <CopyToOutputDirectory>Always</CopyToOutputDirectory>
        </None>
    </ItemGroup>
```

```
</Project>
```

Note: In the future, the PluginBase and Shared references will be replaced by NuGet packages.

The **plugin.json** metadata file is a simple JSON file that contains some general information about the plugin. It should look like this:

```
{
    "name": "Test Plugin",
    "version": "1.0.0",
    "description": "This is a test plugin",
    "author": "Your Name",
    "dependencies": []
}
```

Plugin Class

Create a new class that implements the `IPlugin` interface. Alternatively, one can also make use of the `PluginBase` class that provides some default implementations.

```
public class TestPluginClass : PluginBase<TestPluginClass>
{
    protected override void Initialize()
    {

    }

    protected override void BuildEndpoints(IEndpointRouteBuilder routeBuilde
r)
    {

    }

    protected override void ConfigureServices(IServiceCollection services)
    {
```

```
    }

    protected override void SystemStart()
    {

    }
}
```

Additional Considerations

- Make sure to include all static assets in the output folder of the plugin.
- Make sure there are now overlapping .dll file dependencies with the backend application for shared types.
- Make sure to include all necessary files in the .csproj file.
- Make sure to include all necessary dependencies in the `plugin.json` file.

Developing plugins locally

Building the plugin

To build a plugin, you need to have the following prerequisites:

- .NET 9 SDK
- A source build of the Mpm-Smart Controller
- The current version of the `PluginBase.dll` and `Shared.dll` libraries

Put your plugin dll output in the plugin directory of the controller application. Hot reloads aren't supported, so you have to restart the backend every time you rebuild the plugin.

Managing Plugin Services

Introduction

All Services are divided into two parts:

- **Core Services**: These are provided by the backend out of the box and can't be modified from the plugin stage
- **Plugin Services**: These are provided by the plugins and can be modified from the plugin. They also offer a way for plugin to communicate with each other.

Logging

The core application and plugin modules use two separate loggers. For development purposes, all logs are uniformly output to the same console. On disk, you will find a log directory with the following files:

- **backend.log**: Contains all logs from the core application layer
- **error.log**: Contains all errors from the core application layer
- **plugin.log**: Contains all logs from the plugin modules
- **plugin-error.log**: Contains all errors from the plugin modules

These log files use a plain text in UTF-8 format and use the following layout: `Date Time Timezone [LEVEL] Message` For example: `2024-11-04 07:55:17.633 +01:00 [INF] Starting Mpm-Smart`

The logging system uses the default Microsoft.Extensions.Logging base library together with serilog for the backend and plugin modules. To use the logger, inject the `ILogger<T>` interface into your service.

Since there logs can also be exported as structured logs, make use of the message templating instead of using string interpolation and pass down the full exception instance where applicable.

Database Management

The core application layer uses Entity Framework Core to manage its databases. You can also use your own custom database models in your plugins, but using Entity Framework Core with SQLite is the only first party supported option. When registering a Database Context in your plugin service collection, its provided migrations will automatically be picked up by the `DbInitializer` and applied to the database on startup. If the database is not present, it will be created.

> ⚠ **Migrations**

When using migrations, make sure to specify the correct migration assembly in a way that is independent of the hosting environment or the startup project.

Seeding

A system to automatically seed plugin databases is not yet implemented. See Project plan for open issue.

Plugins can also access the `SystemDbContext` by using the application service provider provided on initialization of the plugin instance.