Mpm-Smart Documentation

Our very smart smart-home app

Team-Mpm 2024

Table of Contents

Home 2
User Guide 3
Initial Setup 4
User Management 5
Devices Integration 6
Plugin Management 7
Developer Documentation 8
Plugins 9
Creating your first Plugin 10
Developing plugins locally 13
Managing Plugin Services 14

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

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</AppendTargetFrameworkToOutputP
ath>
    </PropertyGroup>

    <ItemGroup>
        <ProjectReference Include="..\plugin_base\PluginBase.csproj"
Private="False"/>
        <ProjectReference Include="..\..\Shared\Shared.csproj"
Private="False"/>
    </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
routeBuilder)
    {

    }
    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

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.