# Introduktion

Du har kanske smarta lampor från Philips i serien Hue som kan styras med Philips app. Skulle det inte vara roligt att kunna visa en juleffekt med lamporna? Denna artikel visar hur det kan åstadkommas med ett program som byggs med Microsofts .NET Core, som nyligen kommit i ny version, 3.0.

.NET Core kan köras på Windows, MacOS och Linux, och på Intel och ARM, vilket betyder i stort sett vilken dator som helst, inklusive Raspberry Pi.

Effekten består i att lamporna går från rött till vitt och tillbaka till rött. Lamporna ska helst vara av Color Ambience-typ, men det går att göra en del med White Ambience också.

Det första beslutet gäller utvecklingsmiljö, och där finns det tre huvudspår:

* Visual Studio, som finns för Windows och MacOS, är den mest bekväma miljön, men också den ”tyngsta”. Ladda ned gratisversionen *Community Edition* från [*https://visualstudio.microsoft.com/*](https://visualstudio.microsoft.com/).
* Din favoriteditor, t.ex. Visual Studio Code, Sublime eller Notepad++, samt .NET Core SDK ([*https://dotnet.microsoft.com/download*](https://dotnet.microsoft.com/download)). Ett bra alternativ för Linux-användare och de som gillar kommandorad.
* Om du bara vill kunna köra koden utan att ladda ned någonting går det att bygga och köra i Docker-containrar (förutsatt förstås att du har Docker). Även detta fungerar på Windows, MacOs och Linux, även Raspberry Pi.

Philips Hue-bryggan har ett REST-baserat API (http och JSON, JavaScript Object Notation), vilket gör det enkelt att utveckla för. Först måste du veta IP-adressen till bryggan. Den går att se i din router, eller genom att gå till [*https://discovery.meethue.com/*](https://discovery.meethue.com/) i en webbläsare. I mitt fall fick jag svaret att den interna adressen är 192.168.1.86. Nu ska vi skapa en användare genom att skicka en http POST till *http://<ip-adress>/api*. Detta kan göras på olika sätt, t.ex. genom att använda Postman eller curl. Det allra enklaste är kanske att använda API-verktyget som finns inbyggt i bryggan på *<http://<ip-adress>/debug/clip.html>*. Ange */api* i URL-fältet och klistra in följande JSON i *Message Body*:

{"deviceType": "HueEffects#MyDevice"}

(Byt *MyDevice* mot namnet på den dator du tänker använda.) För att få skapa en användare måste du först trycka på den stora knappen på bryggan. Tryck sedan på *POST*-knappen i API-verktyget. Svaret ska bli:

[{"success":{"username": "…"}}]

Spara användarnamnet någonstans. Det ska nämligen skickas i alla API-anrop. Allt detta går att läsa på webben, <https://developers.meethue.com/develop/get-started-2/>.

# Hue-klient

Nu är det dags att börja bygga applikationen. Välj *File->New->Project* i Visual Studio eller använd ett kommandoskal och skriv

dotnet new console -n XmasEffect -o XmasEffect

Detta skapar en mapp med två mallfiler, XmaxEffect.csproj, vilket är projektfilen, och Program.cs som är startpunkten för exekveringen.

Vi kan förändra en lampas tillstånd (färg, färgtemperatur etc), genom att göra en http PUT till en URI enligt mönstret /api/lights/<id>. Här måste vi alltså ha lampans identitet, och det kan vi göra genom att hämta alla lampor i en viss grupp, eller den speciella grupp som innehåller alla lampor tillagda till bryggan.

Det är lämpligt att skriva en klass som hanterar detaljerna kring API:et och översätter parametrarna från C# till JSON och tvärtom. Vi skapar därför en ny fil som vi kallar HueClient.cs som tom ser ut så här:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net.Http;

using System.Text;

using System.Text.Json;

using System.Threading.Tasks;

namespace XmasEffect

{

public class HueClient

{

}

}

(Hela den färdiga koden finns tillgänglig på GitHub: <https://github.com/henriko2018/HueEffects/tree/master/XmasEffect>)

Som indata behöver vi användaren som vi skapade tidigare. Vi behöver också skapa en http-klient från ramverket. Detta gör vi klassens konstruktor (klistras in inuti klassen vi nyss skapade):

private readonly string \_user;

private readonly HttpClient \_httpClient;

internal HueClient(string user)

{

\_user = user;

httpClient = new HttpClient();

}

Lampor kan ordnas i grupper. För att hämta lamporna i en grupp gör vi GET av groups/<grupp-id>-resursen:

internal async Task<LightGroup> GetLightGroup(string lightGroupId)

{

var response = await \_httpClient.GetStringAsync(await GetApiUri($"groups/{lightGroupId}"));

var group = JsonSerializer.Deserialize<LightGroup>(response, new JsonSerializerOptions { PropertyNameCaseInsensitive = true });

group.Id = lightGroupId;

return group;

}

Metoden GetApiUri ser ut så här:

private async Task<string> GetApiUri(string resource)

{

if (\_localIp == null)

\_localIp = await DiscoverLocalIp();

return $"http://{\_localIp}/api/{\_user}/{resource}";

}

Som synes använder denna i sin tur DiscoverLocalIp. Se GitHub-länken ovan för implementering av den.

Raden i GetLightGroup som översätter JSON till C#-objekt refererar en klass som jag valt att kalla LightGroup. Denna definieras i en ny fil som vi kan kalla HueModels.cs:

namespace XmasEffect

{

public class LightGroup

{

public string Id { get; set; }

public string Name { get; set; }

public string[] Lights { get; set; }

}

}

Strängarna i Lights-arrayen innehåller lampornas identiteter. För att ta reda på vilka förmågor varje lampa har måste vi hämta dessa. Lägg till följande metod till HueClient:

internal async Task<Light> GetLight(string lightId)

{

var response = await \_httpClient.GetStringAsync(await GetApiUri("lights/" + lightId));

var result = JsonSerializer.Deserialize<Light>(response,

new JsonSerializerOptions { PropertyNameCaseInsensitive = true });

result.Id = lightId;

return result;

}

Vi måste också kunna förändra varje lampas tillstånd. Detta görs med PUT till /api/lights/<light id>/state. Lägg till följande metod till HueClient:

internal async Task SetLightState(string lightId, object state)

{

var content = JsonSerializer.Serialize(state,

new JsonSerializerOptions {PropertyNamingPolicy = new LowerCaseJsonNamingPolicy()});

Console.Out.WriteLine($"Setting light {lightId} state to {content}...");

var response = await \_httpClient.PutAsync(await GetApiUri($"lights/{lightId}/state"),

new StringContent(content, Encoding.UTF8, "application/json"));

if (!response.IsSuccessStatusCode)

throw new ApplicationException($"Failed to set light state. Code: {response.StatusCode}, content: {await response.Content.ReadAsStringAsync()}");

}

LowerCaseJsonNamingPolicy är en klass som ser till att varje namn i JSON skrivs med gemener, vilket Hue-bryggan kräver:

internal class LowerCaseJsonNamingPolicy : JsonNamingPolicy

{

public override string ConvertName(string name)

{

return name.ToLower();

}

}

# Effekt-koden

Nu är det dags att kombinera metoderna i HueClient. Skapa en ny fil och kalla den XmasEffect.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace XmasEffect

{

internal class XmasEffect

{

public bool Continue { get; set; } = true;

private readonly HueClient \_hueClient;

private readonly Options \_options;

public XmasEffect(Options options)

{

\_options = options;

\_hueClient = new HueClient(options.User);

}

}

}

Continue-flaggan kommer vi senare sätta till false för att stoppa effekten. Options, som vi kommer att skicka från Main, innehåller inställningar för användare och med vilken hastighet färgen ska ändras. Vi lägger nu till en metod att anropa för att starta och köra effekten. Den hämtar först lamporna i grupp 0, som är en speciell grupp som innehåller alla lampor. Sedan går vi igenom alla lampor och ser vilka som har förmåga att ändra kulör och mättnad, eller åtminstone färgtemperatur. Sedan sätter vi på lamporna med lämpliga parametrar och sist kör vi effekten.

internal async Task Start()

{

// Always use the special group "0", which contains all lights. This can of course be changed.

var lightGroup = await \_hueClient.GetLightGroup("0");

var capableLights = await GetCapableLights(lightGroup);

await InitLights(capableLights);

await RunEffect(capableLights);

}

private async Task<(List<Light> colorLights, List<Light> ambienceLights)> GetCapableLights(LightGroup lightGroup)

{

var lights = new List<Light>();

foreach (var lightId in lightGroup.Lights)

{

var light = await \_hueClient.GetLight(lightId);

light.Id = lightId;

lights.Add(light);

}

var colorLights = lights.Where(light => light.Capabilities.Control.ColorGamut != null).ToList();

var ambienceLights = lights.Where(light => light.Capabilities.Control.Ct != null).Except(colorLights).ToList();

Console.Out.WriteLine($"Color light(s): {string.Join(',', colorLights.Select(l => l.Id))}");

Console.Out.WriteLine($"Ambience light(s): {string.Join(',', ambienceLights.Select(l => l.Id))}");

return (colorLights, ambienceLights);

}

private async Task InitLights((List<Light> colorLights, List<Light> ambienceLights) capableLights)

{

Console.Out.WriteLine("Turning on light(s)...");

foreach (var colorLight in capableLights.colorLights)

await \_hueClient.SetLightState(colorLight.Id, new State {On = true, Xy = new[] { 0.675f, 0.322f } }); // Red

foreach (var ambienceLight in capableLights.ambienceLights)

await \_hueClient.SetLightState(ambienceLight.Id, new State {On = true});

}

Här är koden som kör effekten, och ändrar mättnad (saturation) och/eller temperatur. Principen är att i små steg gå från minimal mättnad (vit) till maximal (röd) och sedan åt andra hållet.

private async Task RunEffect((List<Light> colorLights, List<Light> ambienceLights) capableLights)

{

Console.Out.WriteLine("Running effect...");

// Documentation says "sat:25 always gives the most saturated colors and reducing it to sat:200

// makes them less intense and more white" but it is the other way around.

const float maxSat = 200;

const float minSat = 25;

const float maxCt = 454;

const float minCt = 153;

const float timeInterval = 1; // seconds

// Calculate step if we update once per time-interval.

var noSteps = \_options.CycleLength / timeInterval / 2; // Number of steps up or down

var satStep = (maxSat - minSat) / noSteps;

var ctStep = (maxCt - minCt) / noSteps;

while (Continue)

{

var sat = minSat;

var ct = minCt;

// Start at minSat, i.e. white, and go towards maxSat, i.e. red (higher value).

// For ambience lights, we go from minCt, i.e. cold, to maxCt, i.e. warm (higher value).

for (var i = 0; i < noSteps && Continue; i++)

{

await UpdateAndWait(capableLights, sat, ct, timeInterval);

sat += satStep;

ct += ctStep;

}

// Go in the opposite direction, i.e. from red (maxSat) to white (minSat) (lower value).

for (var i = 0; i < noSteps && Continue; i++)

{

await UpdateAndWait(capableLights, sat, ct, timeInterval);

sat -= satStep;

ct -= ctStep;

}

}

Console.Out.WriteLine("Canceled.");

}

private async Task UpdateAndWait((List<Light> colorLights, List<Light> ambienceLights) capableLights, float sat, float ct, float timeInterval)

{

var startTime = DateTime.Now;

// We don't await \_hueClient below, since we're not interested in the result of the operation.

// If it fails, it will probably work next time.

foreach (var light in capableLights.colorLights)

\_hueClient.SetLightState(light.Id, new {sat = (int)sat});

foreach (var light in capableLights.ambienceLights)

\_hueClient.SetLightState(light.Id, new {ct = (int)ct});

var timeToWait = startTime.AddSeconds(timeInterval) - DateTime.Now;

await Task.Delay(timeToWait);

}

# Uppstart

Den sista pusselbiten är nu att anropa Start från Program.Main:

internal static class Program

{

private static async Task Main(string[] args)

{

await Parser.Default.ParseArguments<Options>(args).MapResult(

async options =>

{

var effect = new XmasEffect(options);

Console.CancelKeyPress += (\_, eventArgs) =>

{

Console.Out.WriteLine("Ctrl-C");

eventArgs.Cancel = true;

effect.Continue = false;

};

await effect.Start();

},

errors => Task.CompletedTask);

}

}

internal class Options

{

[Option('u', "user", Required = true, HelpText = "User (hexadecimal string) created with POST /api")]

public string User { get; set; }

[Option('c', "cycle-length", Required = false, Default = 15, HelpText = "Length of cycle in seconds")]

public int CycleLength { get; set; }

}

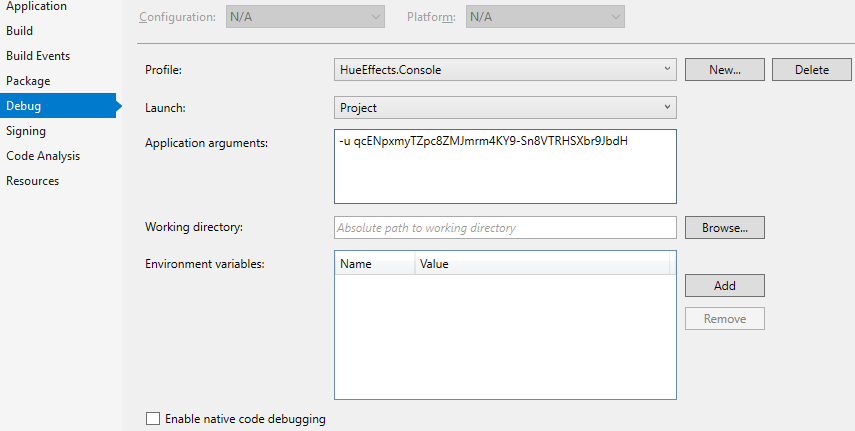
Notera är att vi måste ändra standard-deklarationen av Main, static void Main, till static async Task Main, eftersom vi har en rad med await. Vi tar också hjälp av ett bibliotek, CommandLine, för att enkelt hantera kommandoargument. Detta installeras enkelt genom att i kommandoskalet skriva

cd XmasEffect

dotnet add package CommandLine

Första parametern till MapResult är en funktion som körs om tolkning av kommandoargument lyckas. Här sätter vi upp en lyssnare för händelsen Console.CancelKeyPress, för då behöver vi signalera till XmasEffect.RunEffect att stanna. Den andra parametern till MapResult är en funktion som ska köras om tolkning av kommandoargument misslyckas. Här gör vi ingenting mer än att returnera en redan färdig Task, eftersom bibliotekskoden automatiskt skriver ut hjälptext.

För att köra från Visual Studio, välj egenskaper på projektet, gå till Debug-fliken och fyll i kommandoargumenten. Endast användaren är obligatorisk. Exempel: -u qcENpxmyTZpc8ZMJmrm4KY9-Sn8VTRHSXbr9JbdH.



Tryck sedan F5. Om du inte har Visual Studio, bygg och kör från kommandorad:

dotnet build

.\bin\Debug\netcoreapp3.0\XmasEffect

Eller om du endast hämtat den färdiga koden och vill köra den i Docker, kör BuildWindowsContainer.bat eller BuildLinuxContainer.sh följt av docker run xmas-effect:latest.

Som sagt, koden finns på GitHub, tillsammans med ett större exempel, där två olika effekter kan styras från ett webbgränssnitt.