OULUN AMMATTIKORKEAKOULU

DEVELOPMENT OF A LIVE LOCATION TRACKING MODULE FOR THE MAGIC MIRROR PLATFORM

Riku Partanen
Bachelor's Thesis
Autumn 2024
Degree Programme in Information technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology
Option of Device and Product Design

Author(s): Riku Partanen
Title of thesis: Development of a live location tracking module for the Magic Mirror platform
Thesis supervisor(s): Jouni Juntunen
Term and year of completion: Autumn 2024
Pages: 32 + 1 appendix

This report was written at Oulu University of Applied Sciences as a final thesis. The goal of this thesis was to develop a module for the Magic Mirror environment that would receive and display a smart phone's location in real time. The topic was chosen because a module suited for such needs did not exist on the Magic Mirror third party module list.

The methods used in this thesis were following the documentation of a third-party Magic Mirror module to create a custom module and building a mobile application as an Expo project from a template. The Magic Mirror module consists of a frontend that utilizes Google Maps JavaScript API to render and update a map, and a backend that communicates between the mobile application and the frontend by using Node.js and the built in Magic Mirror notification system. The mobile application uses the Expo framework's built in libraries for mobile device location access and node.js communication.

The result of the thesis was a functioning live location tracking module running on the Magic Mirror application, and a smart phone application that works alongside it. The aim of this thesis was only to develop a functioning proof-of-concept, but the module could be improved upon in various ways by focusing on areas such as the security and scalability of the module. Third party developers are encouraged to publish their own Magic Mirror modules for the public to use, however the module made in this thesis would need more refinements before it could be published for anyone to use because of the aforementioned reasons.

Keywords: Magic Mirror, Node.js, Expo, JavaScript, Google Maps JavaScript API, Raspberry Pi

CONTENTS

1 INTRODUCTION

This thesis documents the development of a live location tracking module for the Magic Mirror environment, and a matching mobile application for sharing the location to the module. The Magic Mirror platform is an open-source modular smart mirror environment that is used to display various modules that present information such as the user's upcoming calendar events, the local weather or news. (Magicmirror 2016.)

The idea for this project came from personal experience. Before I went abroad for my exchange studies, I had an idea to turn my computer display and a spare Raspberry Pi -microcontroller at home into a magic mirror that would display different relevant information about me from different modules, including my exchange city's weather, current time and my live location. However, after researching the Magic Mirror forums, I couldn't find a suitable location tracking module for my needs. A few location-related modules did exist, but they were not aimed for displaying a live location of a smartphone. Instead, they displayed a given area's traffic conditions or planes currently flying above. I scrapped my idea back then due to lack of time and knowledge on the topic, but I was still interested in it. Two years later I needed a topic for my thesis, and I realized I could finally create a solution to the problem I had had earlier.

The goal of my thesis is to develop a module to show a person's location in real time on a Magic Mirror display. To do this I need to create two programs that work together: a Magic Mirror module and a mobile application. My aim is only to develop a functioning proof-of-concept of my idea that meets the above-mentioned criteria. The Magic Mirror developers encourage third-party developers to publish their own modules for the public to use, but publishing your own module requires documentation, polish and upkeep of the module, which are not in the scope of my thesis.

The development tools used in my project are the Expo framework and TypeScript coding language for the mobile application, and JavaScript for the Magic Mirror module. Expo is a framework and platform for building React Native

mobile applications. It can run natively on Android, iOS and as a web application, but I will be only developing and testing it on my own Android smartphone. I chose to use Expo as the framework as it has built-in SDKs that offer APIs for features such as push notifications, camera and more importantly GPS. (Expo Documentation s.a.)

My Magic Mirror module will display the location of the smart phone on a map that will update in real time. The two applications' communication works by the mobile app sending a fetch call, and the module receiving it on an Express server port. I have limited experience with JavaScript and mobile application development as I chose to major in product and device design, so this project will allow me to learn a lot of new and useful areas of information technology.

2 OVERVIEW OF USED TECHNOLOGIES

The Magic Mirror platform is an open-source modular smart mirror platform that works by combining hardware and software to create a display capable of showing information such as time, weather, calendar events, news feeds, and other user-defined content. A typical Magic Mirror project can be seen in figure 1. It was first created by Michael Teeuw in 2014 as a do-it-yourself project to display simple information on a spare screen (Teeuw 2014). Since then, it has received a major overhaul and grown into a popular open-source platform with hundreds of modules of different categories ranging from finance and transport to sports and education. The most common use case of the program is to set up a computer screen displaying the Magic Mirror program behind a one-way mirror, causing the light from the display to pass through. The background of the program is completely black, and all of the displayed modules are white. Because of the high contrast difference and the reflectiveness of the one-way mirror, it looks like the program's modules magically appear on the mirror even when the mirror is reflecting the light shined on it normally, hence the name. (Barnes 2016.)

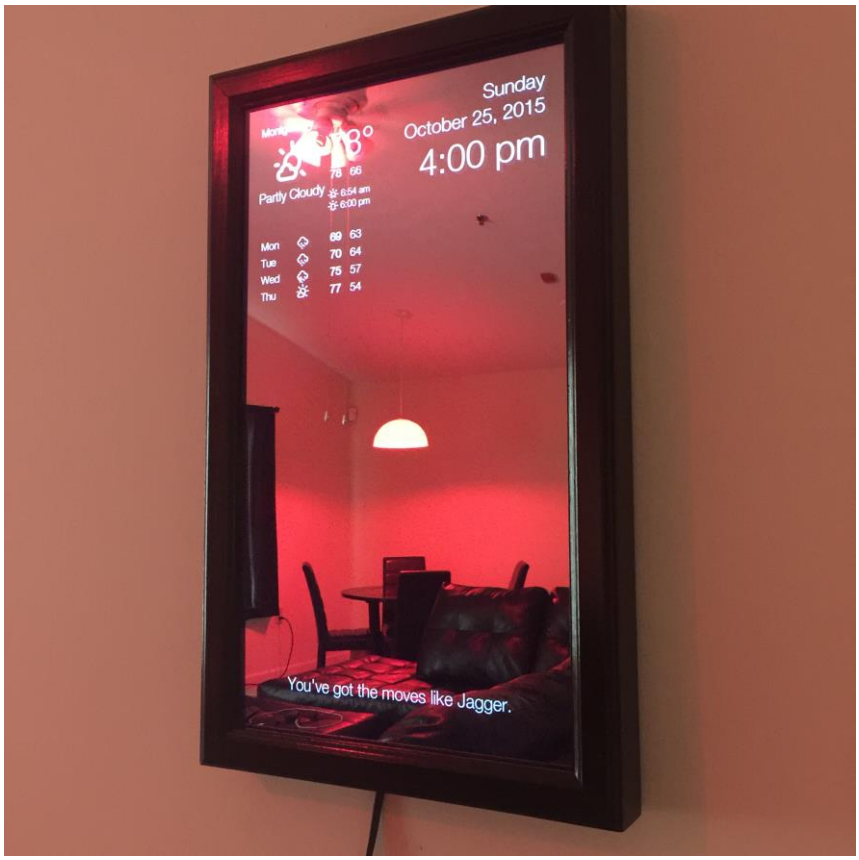


Figure 1. An example of the Magic Mirror environment being displayed on a screen behind a one-way mirror (Barnes 2016).

The core software of the Magic Mirror runs on a lightweight Node.js server, using Electron for rendering (Magicmirror s.a. g). It works as a web application that is displayed locally on the mirror's screen. The Magic Mirror platform is developed to run on a Linux-based environment, specifically on a Raspberry Pi, but it also has a Windows version. (Magicmirror s.a. b.)

2.1 Magic Mirror modules

The concept of the Magic Mirror system relies on the numerous modules. They are small self-contained applications that are made for specific functionalities. The program comes with seven default modules for displaying common features such as a clock, a calendar of public holidays, local weather and a news feed (Magicmirror s.a. e). A default screen of the Magic Mirror application can be seen in figure 2. For any other functionalities, the third-party module wiki has hundreds

of additional modules ranging from sports to education (MagicMirrorOrg s.a.). Modules run independently of each other, but they can also communicate with each other by using the built-in notification system (Magicmirror s.a. f).

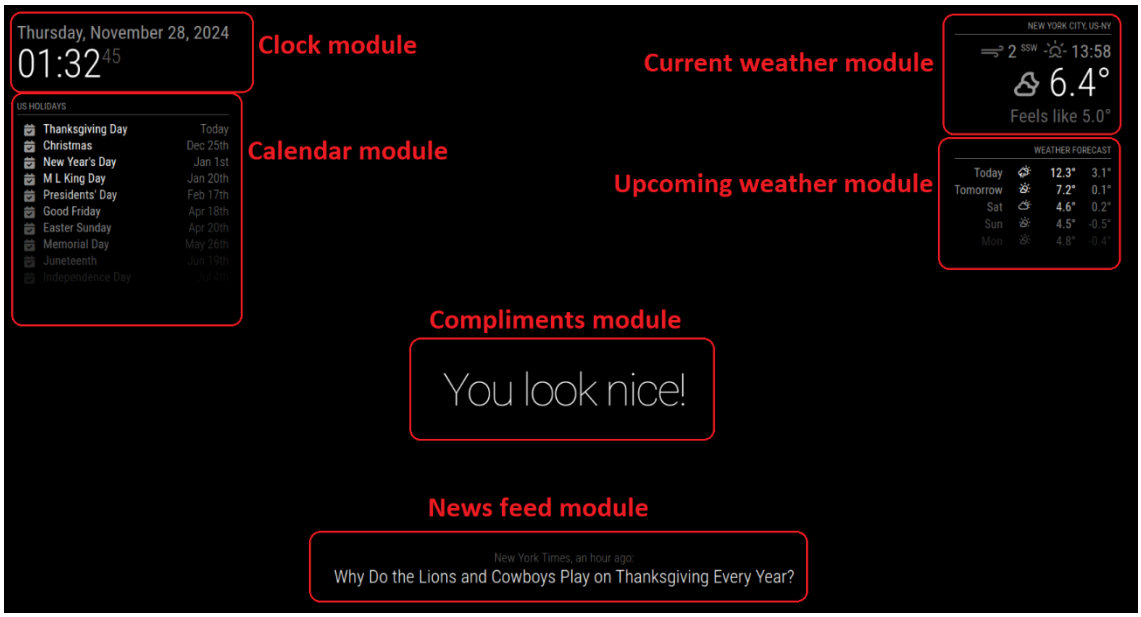
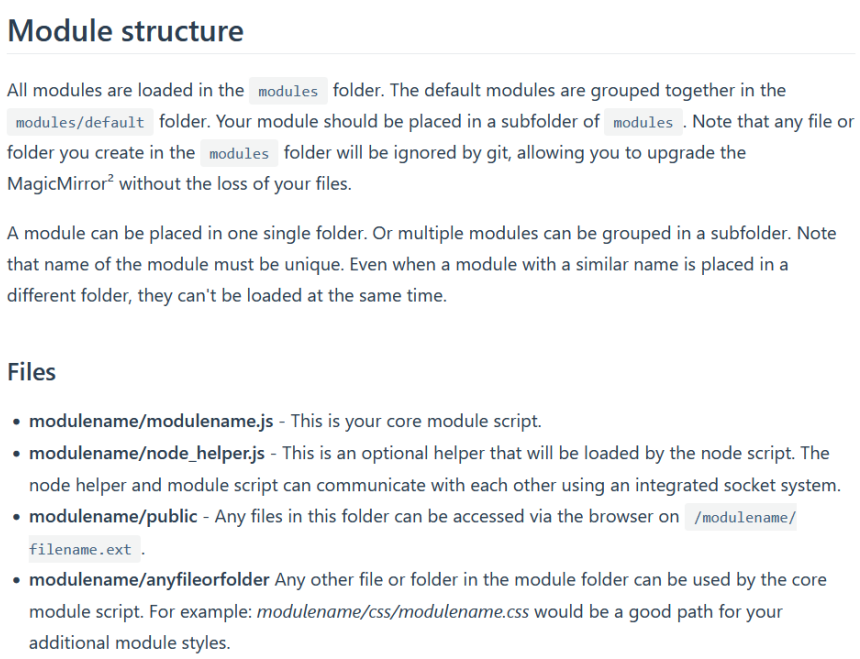


Figure 2. Default view of the Magic Mirror program after installation. Different modules are circled for visualization.

Each module is operated through a configuration file called config.js that is automatically included when installing the Magic Mirror program. By editing this file, the position, size and settings of each module can be specified. A third-party module can be added to the display by cloning it from its GitHub page to the Magic Mirror module folder, and then adding its configuration options to the config.js file. (Magicmirror s.a. c.)

2.1.1 Developing a Magic Mirror module

Most of the Magic Mirror modules are developed and shared by the community. In order for the numerous third-party modules to be understandable and usable by anyone, rules and guidelines are supplied on the Magic Mirror website. Some of these instructions can be seen in figure 3. (Magicmirror s.a. d.)



Module structure

All modules are loaded in the `modules` folder. The default modules are grouped together in the `modules/default` folder. Your module should be placed in a subfolder of `modules`. Note that any file or folder you create in the `modules` folder will be ignored by git, allowing you to upgrade the MagicMirror[2] without the loss of your files.

A module can be placed in one single folder. Or multiple modules can be grouped in a subfolder. Note that name of the module must be unique. Even when a module with a similar name is placed in a different folder, they can't be loaded at the same time.

Files

- **modulename/modulename.js** - This is your core module script.
- **modulename/node_helper.js** - This is an optional helper that will be loaded by the node script. The node helper and module script can communicate with each other using an integrated socket system.
- **modulename/public** - Any files in this folder can be accessed via the browser on `/modulename/filename.ext`.
- **modulename/anyfileorfolder** Any other file or folder in the module folder can be used by the core module script. For example: *modulename/css/modulename.css* would be a good path for your additional module styles.

Figure 3. A screenshot of module development documentation from the Magic Mirror website (Magicmirror s.a. d).

A module template that includes a basic module structure is also provided on the Magic Mirror documentation website. It showcases some of the basic features that can be used when developing a custom module. (Rosenbaum 2024.)

A custom module can have variables that can be edited from the config.js file. For example, if a module needs an API key to function, instead of needing to manually insert the key to a module's own code files, it can be put into the config.js file as a variable that the modules can read. (Magicmirror s.a. c.)

2.1.2 Structure of a Magic Mirror module

A magic Mirror module consists of several files and components. The main JavaScript file named after the module handles the frontend side of the module, rendering the information displayed with the Electron-based browser. On top of the main file, a file called node_helper.js can be included to handle the backend data processing and communication with other modules and services. Unlike the frontend file, the backend file can use functions like require and express. The frontend and backend logic are separated into different files, making it easier to

debug and define functionality between them. The Magic Mirror program automatically handles the linking between the main frontend and the node_helper.js backend file when they exist in the same directory. A config.js file must also be included in a module. This file is used for configuring the size, location and settings of a module. (Magicmirror s.a. h.)

Modules can include a CSS file for custom stylesheets or animations to define their appearance. The Magic Mirror program also supports HTML templates that can be used for structuring the visual output. (Magicmirror s.a. a.)

2.2 Communication of the Magic Mirror program

Communication within the Magic Mirror program occurs between the modules and the main core system through a notification system. Modules can send and receive notifications using the built-in notificationReceived() and sendNotification() commands. These commands can also be used to communicate between a module's frontend and backend, or between different modules. For example, a weather module could send a notification including temperature data to another module to display in a different format. (Magicmirror s.a. f.)

The modules can also communicate with other devices over the internet. For example, a public holiday calendar module can fetch the upcoming holidays from an external API. The Magic Mirror program can also be configured for external access. By enabling remote connections, users can view and configure their Magic Mirror's display from other devices. The program can be launched in server mode, where it doesn't open a new window to display the modules on the device, but instead host the whole system on a local network website so that the program can be displayed on another screen wirelessly. (Magicmirror s.a. b.)

How widely other devices are able to access the Magic Mirror can be configured in the config.js file. The default setting is set to not accept any connection requests from outside localhost, but this can be set to allow local area network devices, only whitelisted IP-addresses or anyone. (Magicmirror s.a. b.)

2.3 Hardware for the Magic Mirror

While the Magic Mirror software is fairly light and uses little memory, it requires specific hardware components to function effectively. The program is developed to run on a Raspberry Pi microcontroller, which the developers recommend using. Other hardware can also be used, but new versions of the program are only tested on a Raspberry Pi. The Raspberry Pi microcontrollers are also physically small and require very little power to run, so they are ideal for hiding behind a thin mirror. The Magic Mirror environment runs by using Electron as the app wrapper, which only supports the Raspberry Pi 2 and more advanced models. (Magicmirror s.a. g.)

In addition to the microcontroller, the Magic Mirror system requires a computer screen to display its contents. The screen serves as the primary interface for showing the modules' contents. Typically, users repurpose an old monitor or purchase a small, lightweight display that fits behind the two-way mirror, to which the computer running the Magic Mirror program connects via an HDMI or DisplayPort cable. An example of a hardware setup used for a Magic Mirror project, containing a Raspberry Pi 3B and a computer screen, can be seen in figure 4.

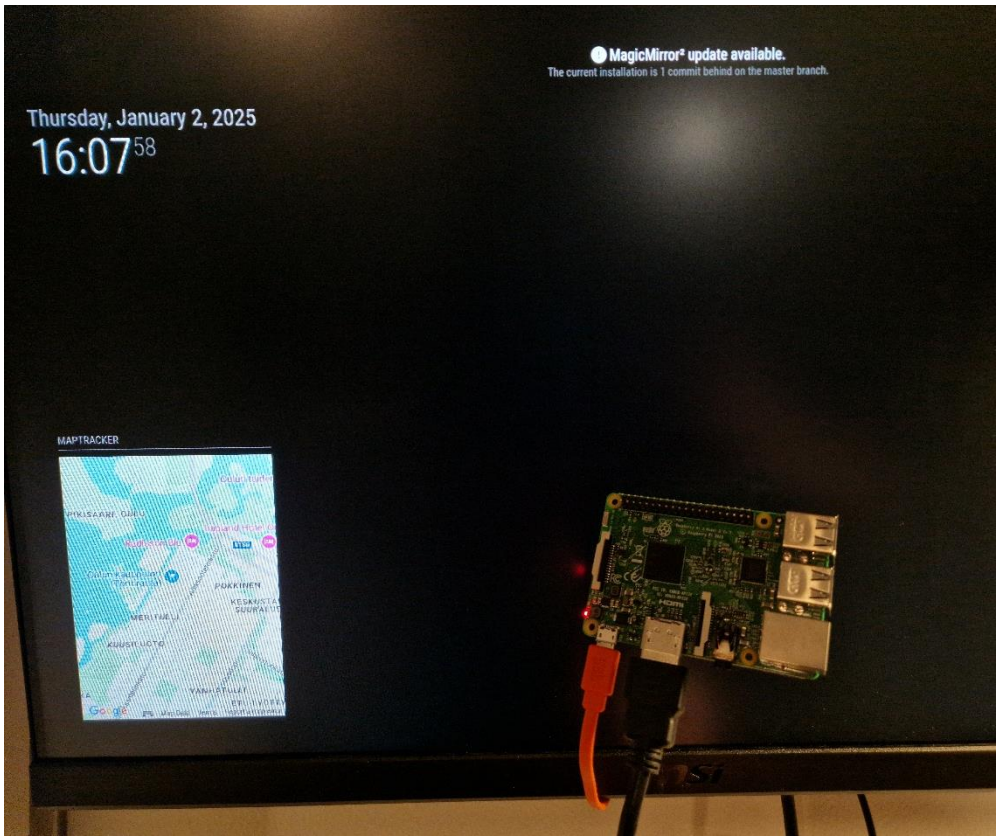


Figure 4. Hardware setup used in the project. A Raspberry Pi 3B running the Magic Mirror program is connected to a computer display via a USB and an HDMI cable.

Additionally, the Magic Mirror program can be configured to run on a local port, allowing it to be accessed by other devices on the same network. This feature enables users to display the Magic Mirror interface on alternate devices such as tablets, smartphones, or PCs, either as a supplementary screen or as the primary display for remote control purposes. (Magicmirror s.a. b.)

2.4 Expo framework

Expo is a popular open-source platform for building React Native mobile applications for Android, iOS and web apps. It provides tools and built-in APIs for developing, deploying and managing apps. Because the Expo framework has cross-platform development, all the written code runs natively on all platforms. (Expo documentation s.a.)

One of the main features of Expo is its testing and debugging process through the Expo Go app. This mobile application allows users to test projects quickly and easily without compiling all the code into an installable file every time a small change is done. Users can scan a QR code generated by the Expo development server, which loads the app on the mobile device. (Moedano 2024.)

Expo's numerous built-in APIs provide straightforward access to device cameras, GPS, notifications, and sensors. These APIs simplify the integration of functionalities, allowing the user to focus on building application-specific features instead of implementing lower-level device interactions. (Expo documentation s.a.)

2.5 Google Maps JavaScript API

Google Maps JavaScript API can be used to display Google Maps in web applications. An example of this can be seen in figure 5. As it is cheap, easy to use and well-documented, it was chosen to be used in this project. The displayed map's functionality and appearance can be configured to fit the user's needs. Users can adjust settings such as map type, zoom levels, and controls to create a custom map made for specific requirements. (Google Developers s.a. b.)

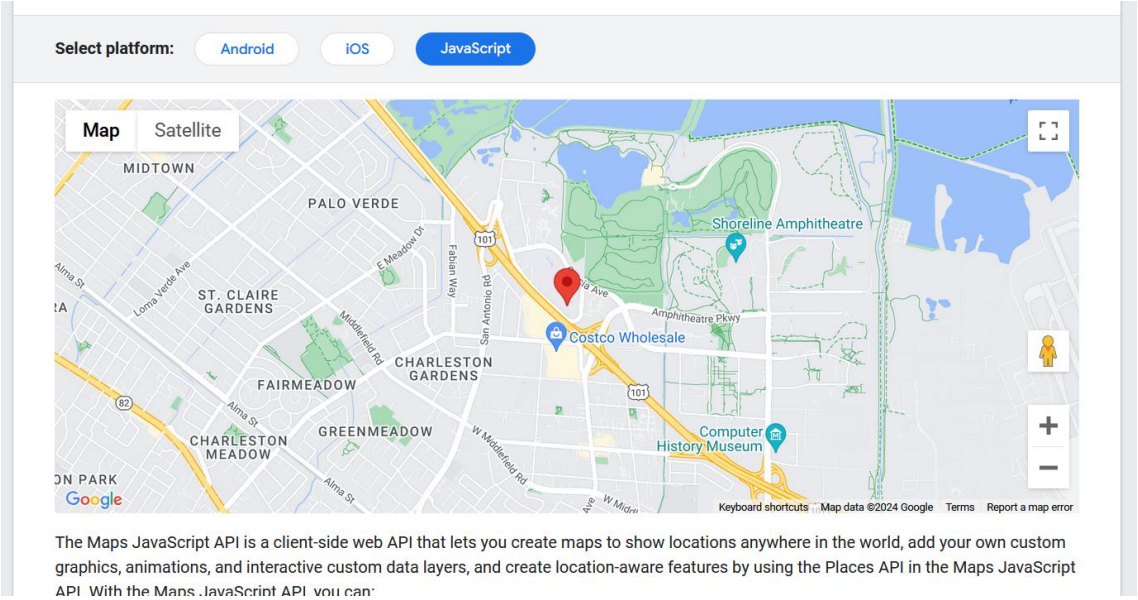


Figure 5. An example of a map displayed on a webpage using Google Maps JavaScript API (Google Developers s.a. b).

The API also provides built-in markers that can be used to indicate locations on the map. These markers are customizable, allowing users to set their position, change their appearance, and add labels or pop-up information windows. For example, in this project, a marker was used to showcase a person or device's location and dynamically update the map to center on the marker, so that the relevant area is always shown. (Google Developers s.a. a.)

3 IMPLEMENTATION OF THE MODULE AND THE APPLICATION

The finished implementation of my Magic Mirror module and mobile application work together to display the mobile phone's location on the Magic Mirror display. Figure 6 shows the Magic Mirror running with my custom module, the terminal of the Magic Mirror program and a screen capture of the mobile application sharing its location. My module sends a log message to the Magic Mirror terminal every time it receives a new location from the mobile application. These log messages can be seen in the image.

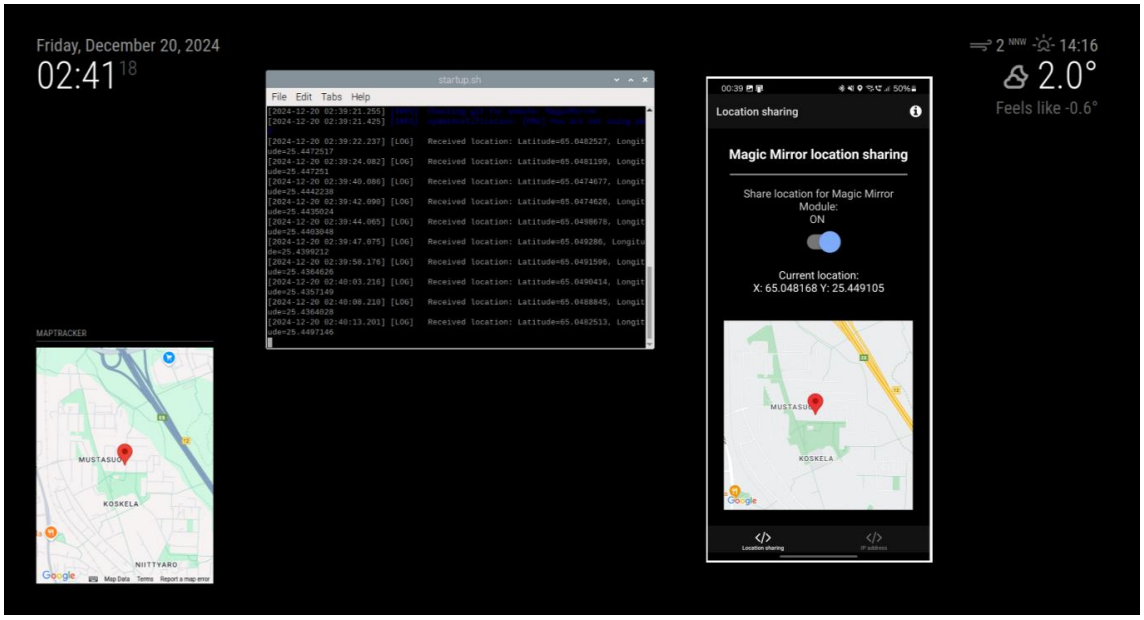


Figure 6. An image of the custom module and the mobile application communicating.

I am running the Magic Mirror program on a Raspberry Pi 3 model B microcontroller. I was originally going to develop it on a Raspberry Pi Zero 2 W, but the Magic Mirror environment runs by using Electron as the app wrapper, which only supports the Raspberry Pi 2 and more advanced models. The device is connected to a computer screen with two cables: an HDMI cable to send the visual data for the screen to display, and a USB cable to provide the microcontroller with electricity to run.

The module consists of two main files: MMM-MapTracker.js and node_helper.js. The former contains the module's frontend functionalities while the latter handles the backend communication. The module's configuration settings include a variable to which a Google Maps API key must be given, an Express server port setting and a map zoom level setting. These settings must be added to the Magic Mirror's main config.js file in order for the module to function properly.

3.1 Module's frontend

The frontend of my module is located in the main file, MMM-MapTracker.js. It is used to render and display the map from Google Maps JavaScript API, and to update the marker in the center of it. The entirety of the frontend code can be seen in figure 7.

There were a few other mapping API options for displaying the map, but I ended up using Google Maps JavaScript API due to its ease of use and thorough documentation. When acquiring the API key, a Google account must be made, and a credit card must be registered to the account. Using Google Maps JavaScript API costs money whenever the API key is used. The module loads the map only once when it is launched, which ends up costing 0.007 USD. Google Maps also has native markers built into the JavaScript API which I can use to display the location of the smart phone and center the map around it. (Google for Developers 2024.)

```
Module.register("MMM-MapTracker", {
  defaults: {
    googleMapsApiKey: "<API_KEY_HERE>", // Replace with your API key
    zoom: 14,
    port: 8081, // Port for receiving fetch requests
  },

  start: function () {
    this.currentLocation = null;
    this.map = null;
    this.marker = null;

    // Load Google Maps JavaScript API
    const script = document.createElement("script");
```

```
    script.src =
"https://maps.googleapis.com/maps/api/js?key="+googleMapsApiKey;
    script.async = true;
    script.defer = true;
    script.onload = () => this.initializeMap();
    document.body.appendChild(script);

    // Start backend listener
    this.sendSocketNotification("START_LISTENER", this.config.port);
  },

  initializeMap: function () {
    this.map = new google.maps.Map(this.dom, {
      zoom: this.config.zoom,
      zoomControl:false,
      streetViewControl:false,
      scaleControl:false,
      rotateControl:false,
      panControl:false,
      mapTypeControl:false,
      fullscreenControl:false,
      center: { lat: 65.013325, lng: 25.464826 }, // Default center
    });
  },

  socketNotificationReceived: function (notification, payload) {
    if (notification === "LOCATION_UPDATE") {
      const { latitude, longitude } = payload;

      // Update map center
      const newLocation = new google.maps.LatLng(latitude, longitude);
      this.map.setCenter(newLocation);

      // Add or update marker
      if (this.marker) this.marker.setMap(null); // Remove old marker
      this.marker = new google.maps.Marker({
        position: newLocation,
        map: this.map,
        title: "Current Location",
      });
    }
  },

  getDom: function () {
    const wrapper = document.createElement("div");
    wrapper.style.width = "300px";
    wrapper.style.height = "400px";
    this.dom = wrapper;
    return wrapper;
```

```
  },
});
```

Figure 7. The frontend code of the custom Magic Mirror module.

The code can be broken down into smaller pieces, with each one handling their own task. The first part of the code, seen in figure 8, handles the launch and setup of the module. The Magic Mirror program launches the module with the given configuration settings found in the config.js file. If the configuration settings in this file are not specified, the default options from the frontend file are used instead. A JavaScript Google Maps map is also initialized by using the API with variables for the current location, map and marker set as null.

```
Module.register("MMM-MapTracker", {
  defaults: {
    googleMapsApiKey: "<API_KEY_HERE>", // Replace with your API key
    zoom: 14,
    port: 8081, // Port for receiving fetch requests
  },

  start: function () {
    this.currentLocation = null;
    this.map = null;
    this.marker = null;

    // Load Google Maps JavaScript API
    const script = document.createElement("script");
    script.src =
"https://maps.googleapis.com/maps/api/js?key="+googleMapsApiKey;
    script.async = true;
    script.defer = true;
    script.onload = () => this.initializeMap();
    document.body.appendChild(script);
```

Figure 8. Code handling the startup and configuration of the module's frontend.

The next part of the code, seen in figure 9, simply sends a notification to the backend file via the port established in the configuration options. The notification called "START_LISTENER" causes the backend file to wake up and start the fetch server, which is used for receiving the location sent by the mobile application.

```
    // Start backend listener
    this.sendSocketNotification("START_LISTENER", this.config.port);
  },
```

Figure 9. Code for sending the wake-up notification for the backend code.

Following that is the code for configuring the map displayed on the Magic Mirror program. This part of the code can be seen in figure 10. Google Maps JavaScript API allows the user to customize the displayed map with various options, for example by enabling or disabling zooming, panning or rotating. In this module most of these features are disabled to achieve a simplified look, which fits the minimalistic aesthetic of the Magic Mirror program better.

```
  initializeMap: function () {
    this.map = new google.maps.Map(this.dom, {
      zoom: this.config.zoom,
      zoomControl:false,
      streetViewControl:false,
      scaleControl:false,
      rotateControl:false,
      panControl:false,
      mapTypeControl:false,
      fullscreenControl:false,
      center: { lat: 65.013325, lng: 25.464826 }, // Default center
    });
  },
```

Figure 10. The frontend code for customizing the displayed map.

Next part of the code, seen in figure 11, is for receiving notifications from the backend and handling the received data. The socketNotificationReceived() function listens for notifications called “LOCATION_UPDATE”. When this notification is received, the code extracts the notification’s payload containing the new coordinates, deletes the old marker on the map, creates a new one with the updated coordinates and centers the map around it.

```
  socketNotificationReceived: function (notification, payload) {
    if (notification === "LOCATION_UPDATE") {
      const { latitude, longitude } = payload;

      // Update map center
      const newLocation = new google.maps.LatLng(latitude, longitude);
      this.map.setCenter(newLocation);

      // Add or update marker
      if (this.marker) this.marker.setMap(null); // Remove old marker
      this.marker = new google.maps.Marker({
        position: newLocation,
        map: this.map,
        title: "Current Location",
      });
    }
  },
```

Figure 11. Frontend code for receiving notifications from the backend and updating the map marker based on the given data.

Finally, the last part of the frontend code, seen in figure 12, contains the getDom() function, which is required in every module for them to function. It gives necessary information to the Magic Mirror program, such as the dimensions of the module.

```
  getDom: function () {
    const wrapper = document.createElement("div");
    wrapper.style.width = "300px";
    wrapper.style.height = "400px";
    this.dom = wrapper;
    return wrapper;
  },
});
```

Figure 12. The frontend code's last part, containing the getDom() function.

3.2 Module's backend

The backend of my module is located in the node_helper.js file. It handles all of the server-side tasks, such as starting an Express server for receiving the post messages from the mobile application and sending a notification to the frontend

containing the new coordinates. The entirety of the backend code can be seen in figure 13.

```
const NodeHelper = require('node_helper');
const express = require('express');
const cors = require('cors');

module.exports = NodeHelper.create({
  start: function () {
    console.log("Starting node helper for MMM-MapTracker...");
    this.expressApp = null;
  },

  socketNotificationReceived: function (notification, payload) {
    if (notification === "START_LISTENER") {
      this.startServer(payload); // Payload contains the port number
    }
  },

  startServer: function (port) {
    if (!this.expressApp) {
      this.expressApp = express();
      this.expressApp.use(cors()); // Enable CORS
      this.expressApp.use(express.json()); // Parse JSON requests

      // Endpoint to receive location updates
      this.expressApp.post('/location', (req, res) => {
        const { latitude, longitude } = req.body;

        if (latitude && longitude) {
          console.log(`Received location: Latitude=${latitude},
Longitude=${longitude}`);

          // Send location to frontend
          this.sendSocketNotification("LOCATION_UPDATE", { latitude,
longitude });
          res.status(200).send('Location received successfully.');
        } else {
          res.status(400).send('Invalid location data.');
        }
      });

      // Start listening on the specified port
      this.expressApp.listen(port, () => {
        console.log(`MMM-MapTracker server is listening on port
${port}`);
      });
    }
```

```
  },
});
```

Figure 13. The entirety of the module's backend code.

The backend code also consists of different parts doing different tasks. The first half of the code is for importing necessary libraries, receiving the code startup notification from the frontend, and establishing and starting the Express server. This part of the code can be seen in figure 14. The required libraries are NodeHelper, Express and CORS. The first of these libraries, NodeHelper, is a library within the Magic Mirror framework that handles the communication and functionality between a frontend and backend file. Express is a light and flexible Node.js web application framework that is used for HTTP communication between the module and the mobile application. Finally, CORS, short for Cross-Origin Resource Sharing, allows the module's backend to accept requests from external origins, such as the mobile application.

```
const NodeHelper = require('node_helper');
const express = require('express');
const cors = require('cors');

module.exports = NodeHelper.create({
  start: function () {
    console.log("Starting node helper for MMM-MapTracker...");
    this.expressApp = null;
  },

  socketNotificationReceived: function (notification, payload) {
    if (notification === "START_LISTENER") {
      this.startServer(payload); // Payload contains the port number
    }
  },

  startServer: function (port) {
    if (!this.expressApp) {
      this.expressApp = express();
      this.expressApp.use(cors()); // Enable CORS
      this.expressApp.use(express.json()); // Parse JSON requests
```

Figure 14. First part of the backend code, used for starting up the backend code and the Express server.

The second half of the code is used to receive the messages from the mobile application and passing the new coordinates to the frontend code. This code can be seen in figure 15. The Express server starts listening for post messages sent to its IP address, specifically to the port number set in the configuration settings with “/location” at the end. For example, this address could be “http://192.168.1.1:8081/location”. If a notification sent to this address contains a payload with variables called “latitude” and “longitude”, these variables are then sent to the frontend with a notification called “LOCATION_UPDATE”.

```
      // Endpoint to receive location updates
      this.expressApp.post('/location', (req, res) => {
        const { latitude, longitude } = req.body;

        if (latitude && longitude) {
          console.log(`Received location: Latitude=${latitude},
Longitude=${longitude}`);

          // Send location to frontend
          this.sendSocketNotification("LOCATION_UPDATE", { latitude,
longitude });
          res.status(200).send('Location received successfully.');
        } else {
          res.status(400).send('Invalid location data.');
        }
      });

      // Start listening on the specified port
      this.expressApp.listen(port, () => {
        console.log(`MMM-MapTracker server is listening on port
${port}`);
      });
    }
  },
});
```

Figure 15. Backend code for receiving location updates and passing them to the frontend.

3.3 Mobile application

The mobile application for providing the user's coordinates to the Magic Mirror module was built from an Expo application template. It features two tabs, one for turning the location sharing on and off, and the other for setting the Raspberry Pi's IP address as the address to send the coordinates to. The first tab also has a Google Maps JavaScript map displaying the user's current location.

The main functionality of the application is to subscribe to the mobile device's location API and send its data to the Magic Mirror module. Most of these functionalities are coded in the main tab's code file. As the file is over 150 rows long and consists of lots of typical formatting code only used for the visual look of the application, only the main functionalities of the code will be elaborated here. The entirety of this file can be seen in appendix 1.

The Expo application uses the location and task manager plugins to function. The location plugin is used to subscribe to the device's GPS coordinates, and the task manager plugin enables the application to send the coordinates to the Magic Mirror as POST messages. The task manager plugin also enables the application to access the GPS API even when the application is not in use, however when the application is running in Expo Go this feature does not work. Google Maps JavaScript API is also used for displaying the map on the app, similar to the Magic Mirror module. However, unlike the module, the mobile application does not need a user generated API key, because it is able to use the key that comes pre-equipped with the Android operating system.

The code for receiving the location data can be seen in figure 16. It is configured to use the highest possible location accuracy of the device in order to receive precise location information. The code is also configured to only receive new coordinates after the device has differentiated a total of 10 meters from the previously received coordinates. This is so that the application is not constantly accessing the GPS API and uses less data and battery. When the location sharing option is turned on in the mobile application, it starts checking the device's GPS coordinates, and when new coordinates are received, they are written over the old ones in the variables "latitude" and "longitude".

```
// Listen to location updates from the background task
const subscription = Location.watchPositionAsync(
  { accuracy: Location.Accuracy.High, distanceInterval: 10 },
  (location) => {
    const { latitude, longitude } = location.coords;
    handleNewLocations([{ coords: { latitude, longitude } }]);
  }
);
```

Figure 16. Mobile application code for accessing the GPS location of the mobile device.

The new coordinates are sent to the Magic Mirror module by the sendLocationToServer() function. This function's code can be seen in figure 17. It sends the information as a POST call to the IP address previously set by the user in the application's second tab. The fetch API requires the 'body' property of a POST request to be a string when using a JSON header. Because of this, the JSON.stringify() function is used to convert the variables "latitude" and "longitude" into a JSON string that can be included in the HTTP request. The code also logs the location sending events into the console differently depending on whether they were successful or not.

```
// Send location to server
const sendLocationToServer = async (latitude: number, longitude:
number) => {
  try {
    const response = await fetch(`${magicMirrorIP}/location`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        latitude,
        longitude,
      }),
    });
    if (!response.ok) {
      console.error('Failed to send location:', response.statusText);
    } else {
      console.log('Location sent successfully!');
    }
  } catch (error) {
    console.error('Error sending location:', error);
  }
};
```

Figure 17. Mobile application's code for sending updated coordinates to the Magic Mirror module.

I am running the mobile application on my phone on the Expo Go app. A screenshot of the main tab can be seen in figure 18. While a complete built

Android app would be preferred in real world usage, the Expo Go app is easier for testing the features and proving the functionalities work.

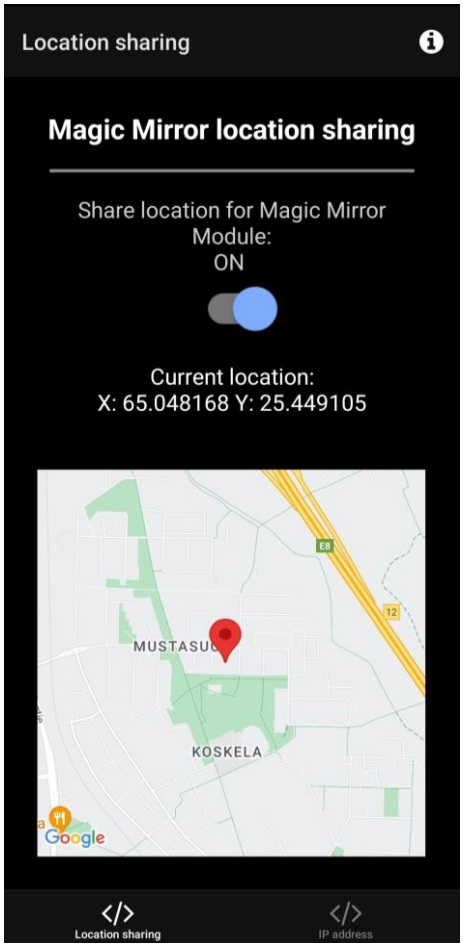


Figure 18. Mobile application default screen, running in the Expo Go app.

4 DISCUSSION

The finalized project functions in a way that it was meant to. It achieves all of the objectives set out for it, albeit more as a proof-of-concept than a polished product ready to be shipped out for anyone to use. I am personally satisfied with the result, but I do want to improve some features to possibly publish it online at some point.

Choosing a project topic that uses a coding language I was not familiar with allowed me to learn a lot more compared to if I had stuck to easier topics I already had experience with. JavaScript, Node.js, Express and Expo were completely alien topics for me before beginning the project but now I would consider myself to be able to use them quite well.

Writing a thesis on this topic was somewhat difficult because most of my sources were from either the official Magic Mirror or Expo documentation websites. Because of this my sources list is shorter than I would have liked it to be, and a lot of the links are subpages under the same website.

Most of the modules on the Magic Mirror website are made by third party individuals who have submitted their works to the Magic Mirror developers to be made public. I would like to publish my project as well, however there are a lot of things I would want to improve on before doing so. The scope of my thesis was only to develop a functioning proof of concept of my topic, and because of this I did not focus on important factors that would need attention before the module could be used by anyone.

The Magic Mirror program can be configured to communicate only within the local network, or with anyone online. During my testing I only allowed the Raspberry Pi to communicate with local devices, because currently my module listens to any fetch request sent to the open Express port without any verification of the sender. This is obviously a security risk and would need improvements if it were to be used continuously. The module could be improved to, for example, use HTTPS instead of HTTP, and constantly update the verification token of the client to keep unnecessary parties away.

I developed the mobile app to be run on my own mobile phone, a Samsung Galaxy S22+. It is only a single device in a market full of different mobile phone brands, models, operating systems and screen sizes. Because I programmed my mobile application in the Expo environment, the application could run on different platforms natively. However, I did not do any testing on other devices or platforms other than my own phone, so the application would likely not scale very well on different screens.

If I were to continue developing this project afterwards, I would figure out some other platform than the Raspberry Pi 3B to program my module on. When working on the module, I often had numerous programs running at the same time, such as several terminals, a development environment, a VNC connection to my laptop and so on. Because of this, the small microcontroller kept overheating and slowed down my progress.

I started building the mobile application from an application template provided by the Expo framework. This helped me setting up the structure and features of the application, but it likely also added lots of unnecessary files and features. The application's purpose is simple and straightforward, but when looking at all of the included files in the project it would not seem that way. The application could definitely be improved by studying exactly which files are necessary for the application to work and which ones could be removed.

The visual look of my module and application are quite simple, which I think fit the minimalistic look of the Magic Mirror. However, I think there are several additional useful features that could be added in the future. One of these is the usage of the markers provided by Google Maps JavaScript API. My module uses them in a very simple way: only display one marker at a time, deleting the old one when inserting a new one. Instead of this the module could have a location history feature, showing the past locations the mobile device has been to. The markers could also be used to set permanent markers for important locations such as home or work, and whenever the mobile device would be close to these important locations the module could display a text saying, "User is at home". Being able to manage these markers from the mobile application would also be a positive improvement.

All in all, while there are numerous ways the end result of my project could be improved and expanded upon, I am satisfied with what I achieved. I had very limited time to complete my project, but I managed to finish everything on time. I believe my finished product could even be useful to others in the future if I decide to publish it at some point.

REFERENCES

Barnes, R. 2016. Magic Mirror. Search date 11.12.2024. https://magpi.raspberrypi.com/articles/magic-mirror

Expo Documentation s.a. Introduction. Search date 9.12.2024. https://docs.expo.dev/get-started/introduction/

Google Developers s.a. a. Markers overview | Maps JavaScript API. Search date 19.12.2024. https://developers.google.com/maps/documentation/javascript/advanced-markers/overview

Google Developers s.a. b. Overview | Maps JavaScript API. Search date 19.12.2024. https://developers.google.com/maps/documentation/javascript/overview

Google for Developers 2024. Google Maps Platform pricing. Search date 22.12.2024. https://developers.google.com/maps/billing-and-pricing/pricing#dynamic-maps

Magicmirror 2016. MagicMirror². Search date 11.12.2024. https://magicmirror.builders/

Magicmirror s.a. a. Custom CSS | MagicMirror² Documentation. Search date 12.12.2024. https://docs.magicmirror.builders/modules/customcss.html

Magicmirror s.a. b. Installation & Usage | MagicMirror² Documentation. Search date 15.12.2024. https://docs.magicmirror.builders/getting-started/installation.html#manual-installation

Magicmirror s.a. c. Introduction | MagicMirror² Documentation. Search date 15.12.2024. https://docs.magicmirror.builders/configuration/introduction.html

Magicmirror s.a. d. Module Development Documentation | MagicMirror² Documentation. Search date 17.12.2024. https://docs.magicmirror.builders/development/introduction.html#general-advice

Magicmirror s.a. e. Modules | MagicMirror² Documentation. Search date 14.12.2024. https://docs.magicmirror.builders/modules/introduction.html

Magicmirror s.a. f. Notifications | MagicMirror² Documentation. Search date 17.12.2024. https://docs.magicmirror.builders/development/notifications.html

Magicmirror s.a. g. Requirements | MagicMirror² Documentation. Search date 9.12.2024. https://docs.magicmirror.builders/getting-started/requirements.html#hardware

Magicmirror s.a. h. The Node Helper | MagicMirror² Documentation. Search date 9.12.2024. https://docs.magicmirror.builders/development/node-helper.html#available-module-instance-properties

MagicMirrorOrg s.a. 3rd Party Modules. Search date 10.12.2024. https://github.com/MagicMirrorOrg/MagicMirror/wiki/3rd-party-modules

Moedano, B. 2024. Expo Go vs Development Builds: Which should you use? Search date 14.12.2024. https://expo.dev/blog/expo-go-vs-development-builds

Rosenbaum, D. 2024. A template for creating new MagicMirror² modules. Search date 15.12.2024. https://github.com/Dennis-Rosenbaum/MMM-Template

Teeuw, M. 2014. The Idea & The Mirror. Search date 10.12.2024. https://michaelteeuw.nl/post/magic-mirror-part-i-the-idea-the-mirror/

APPENDICES

Appendix 1 Mobile application main tab code

Mobile application main tab code **APPENDIX 1**

```
import React, { useState, useEffect } from 'react';
import { StyleSheet, Switch, Alert } from 'react-native';
import MapView, { PROVIDER_GOOGLE, Region, Marker } from 'react-
native-maps';
import * as Location from 'expo-location';
import { startBackgroundLocation, stopBackgroundLocation } from
'@/components/backgroundLocation';
import { Text, View } from '@/components/Themed';
import { magicMirrorIP } from './two';

export default function TabOneScreen() {
  const [isTracking, setIsTracking] = useState(false); // State for
location tracking
  const [currentCoords, setCurrentCoords] = useState({ latitude:
65.0141168, longitude: 25.4697986 }); // Default map location
  const [region, setRegion] = useState<Region>({
    latitude: 65.0141168,
    longitude: 25.4697986,
    latitudeDelta: 0.01,
    longitudeDelta: 0.01,
  }); // Map region state

  // Function to toggle location tracking
  const toggleSwitch = async () => {
    setIsTracking((previousState) => !previousState);

    if (!isTracking) {
      // Start tracking and send data to server
      await startBackgroundLocation();
      Alert.alert('Tracking Enabled', 'Your location is now being
tracked.');
    } else {
      // Stop tracking
      await stopBackgroundLocation();
      Alert.alert('Tracking Disabled', 'Your location tracking has
been stopped.');
    }
  };

  // Send location to server
  const sendLocationToServer = async (latitude: number, longitude:
number) => {
    try {
      // Not sure if I want to keep the /location at the end
      const response = await fetch(`${magicMirrorIP}/location`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          latitude,
          longitude,
        }),
      });
      if (!response.ok) {
        console.error('Failed to send location:',
response.statusText);
      } else {
```

```
        console.log('Location sent successfully!');
      }
    } catch (error) {
      console.error('Error sending location:', error);
    }
  };

  // Background task to handle new location updates
  useEffect(() => {
    const handleNewLocations = async (locations: any[]) => {
      if (locations.length > 0) {
        const { latitude, longitude } = locations[0].coords;

        // Update state with new coordinates
        setCurrentCoords({ latitude, longitude });
        setRegion((prev) => ({
          ...prev,
          latitude,
          longitude,
        }));

        console.log('New location:', latitude, longitude);

        // Send updated location to the server
        await sendLocationToServer(latitude, longitude);
      }
    };

    // Listen to location updates from the background task
    const subscription = Location.watchPositionAsync(
      { accuracy: Location.Accuracy.High, distanceInterval: 10 },
      (location) => {
        const { latitude, longitude } = location.coords;
        handleNewLocations([{ coords: { latitude, longitude } }]);
      }
    );

    return () => {
      subscription.then((s) => s.remove());
    };
  }, [isTracking]);

  return (
    <>
      <View style={styles.container}>
        <Text style={styles.title}>Magic Mirror location
sharing</Text>
        <View style={styles.separator} />
        <Text
          style={styles.additionaltext}
          lightColor="rgba(0,0,0,0.8)"
          darkColor="rgba(255,255,255,0.8)"
        >
          Share location for Magic Mirror Module:{"\n"}
          {isTracking ? "ON" : "OFF"} {"\n"}
        </Text>
        <Switch
          trackColor={{ false: "#767577", true: "#767577" }}
          thumbColor={isTracking ? "#81b0ff" : "#f4f3f4"}
          onValueChange={toggleSwitch}
          value={isTracking}
```

```
          style={{ transform: [{ scaleX: 2 }, { scaleY: 2 }] }}
        />
        <Text style={styles.additionaltext}>
          {"\n"}Current location:{"\n"}
          X: {currentCoords.latitude.toFixed(6)} Y:
{currentCoords.longitude.toFixed(6)}
        </Text>
      </View>

      <View style={styles.mapContainer}>
        <MapView
          provider={PROVIDER_GOOGLE}
          style={styles.map}
          region={region} // Bind the map's region to state
          onRegionChangeComplete={setRegion} // Optional: Handle user-
driven map changes
        >
          {/* Marker to display current location */}
          <Marker coordinate={currentCoords} title="Current Location"
/>
        </MapView>
      </View>
    </>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'flex-start',
    paddingTop: 30,
  },
  mapContainer: {
    position: 'absolute',
    bottom: 30,
    left: 30,
    height: 350,
    width: 350,
  },
  title: {
    fontSize: 25,
    fontWeight: 'bold',
  },
  additionaltext: {
    fontSize: 20,
    textAlign: 'center',
    marginHorizontal: 50,
  },
  separator: {
    marginVertical: 20,
    height: 3,
    width: '80%',
    backgroundColor: '#888',
  },
  map: {
    ...StyleSheet.absoluteFillObject,
  },
});
```