Mosaico

Project report for the "LAM" course

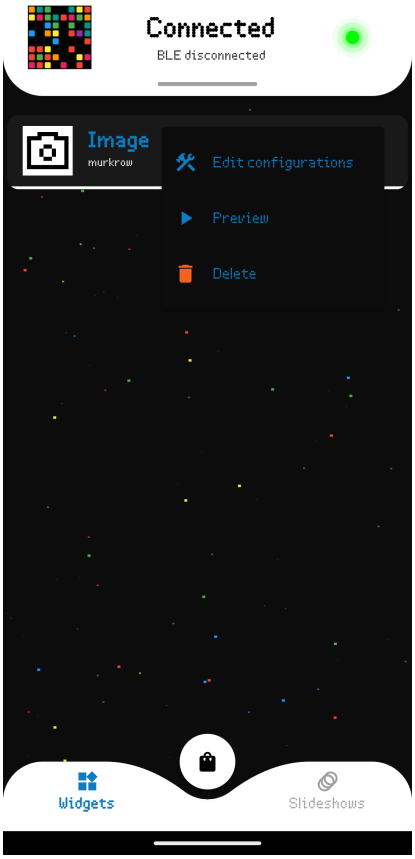
marco.coppola3@studio.unibo.it (0001020433)

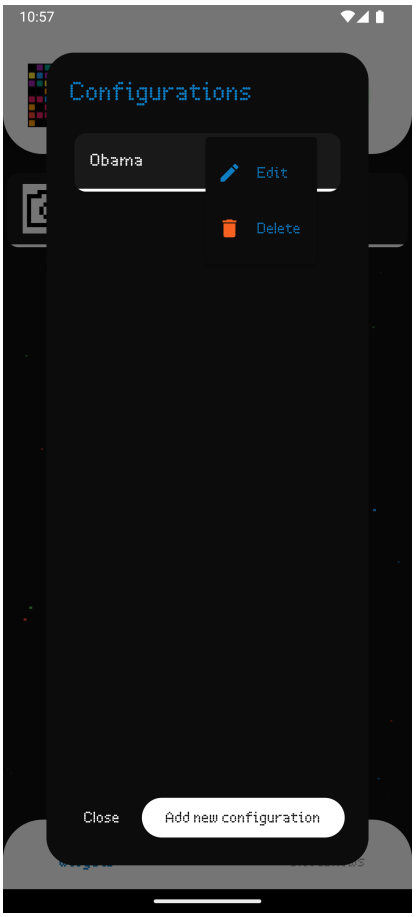
Alma Mater Studiorum Università di Bologna

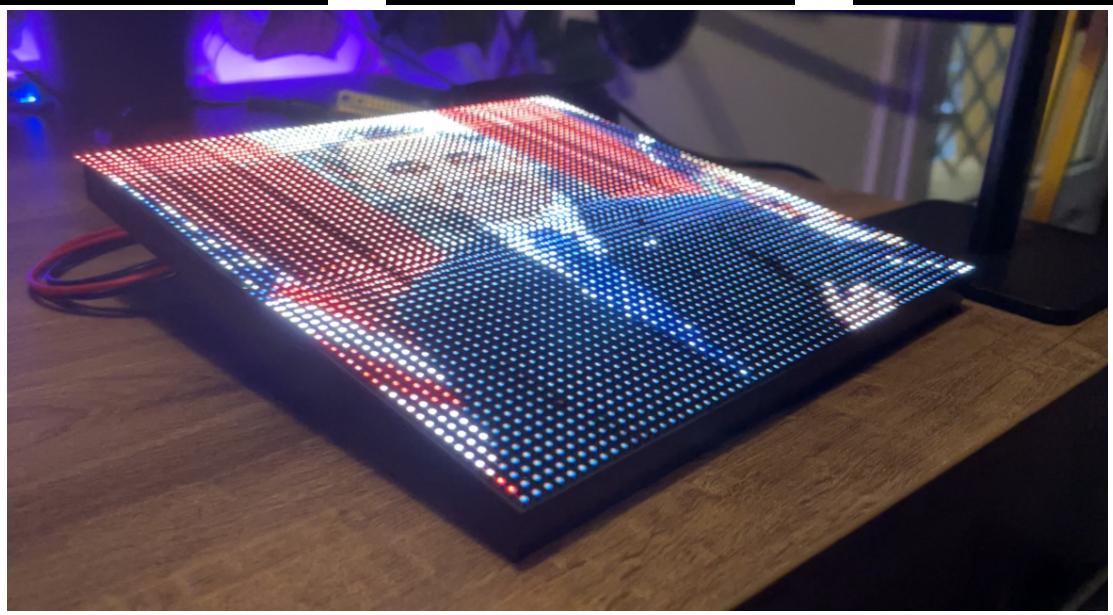
Table of contents

1. Mosaico

Mosaico is a unique (free and open source ❤️) platform that allows **users** and **developers** to create, share, and display custom widgets on a **LED matrix**. This ecosystem is composed of various applications working together to bring vibrant, customizable content to your Raspberry Pi-driven LED matrix.







1.1 Introduction

Mosaico is designed to empower both users and developers by providing an open platform where custom Python widgets can be created and displayed on a LED matrix. Whether you want to show the time, weather, or your latest grocery list, Mosaico makes it easy to develop and deploy your ideas.

1.2 Some examples of widgets

- Display the current time and date.
- Show the weather forecast for your location.
- Create a shopping list widget.
- Upload a custom image and display it as pixel art.
- Write custom text messages or quotes.
- Create animations or visual effects.
- Anything else you can imagine!

1.3 Architecture Overview

The Mosaico Ecosystem consists of:

- Raspberry Pi Software: Written in C++ and Python, this software drives the LED matrix and manages the execution of widgets.
- Mobile App: Developed with Flutter, connects to the Raspberry Pi via BLE and COAP, allowing users to manage widgets, browse the App Store, create slideshows, and configure network settings.
- App Store: Developed with Laravel, a web platform where developers can submit their widgets for others to use.
- IDE: A (dummy) desktop application that allows developers to create and test widgets locally. It is meant to be a lightweight tool to help developers get started with widget development.
- **Simulator**: (part of the software) An X11 window that simulates the LED matrix for development purposes. A web-based simulator is in the works and will allow developers to test their widgets without a physical matrix in the easiest way possible.
- Shared library: A Dart package that contains common code shared between the mobile app and the IDE.

All these project are available on the organization GitHub page.

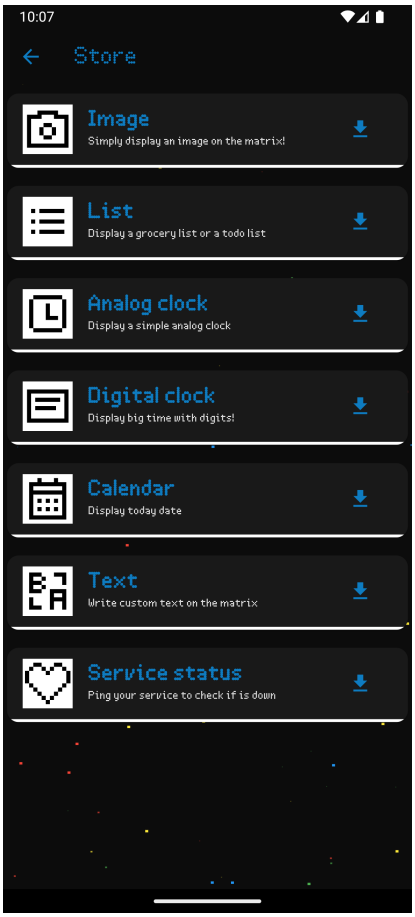
2. Showcase

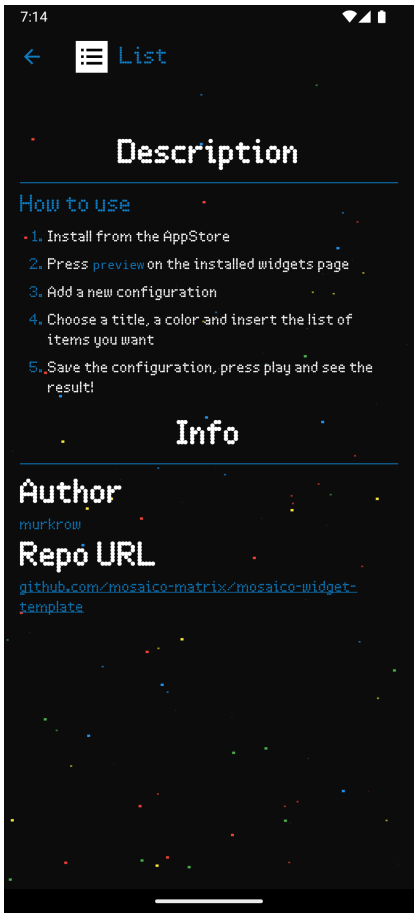
The mobile app interfaces with the Raspberry Pi to:

- Browse and install widgets from the **app store**.
- Configure and manage **installed widgets**.
- Create and manage **slideshows**.
- Check **device status**
- **Control device**

2.1 App Store

The store is the place to find and fall in love with new widgets. Developers can submit their widgets, upload a custom icon, provide a tagline and a rich markdown description. The widget itself is hosted on a **git repo** so that it can be updated and maintained by the developer and easy to retrieve by the Raspberry Pi.

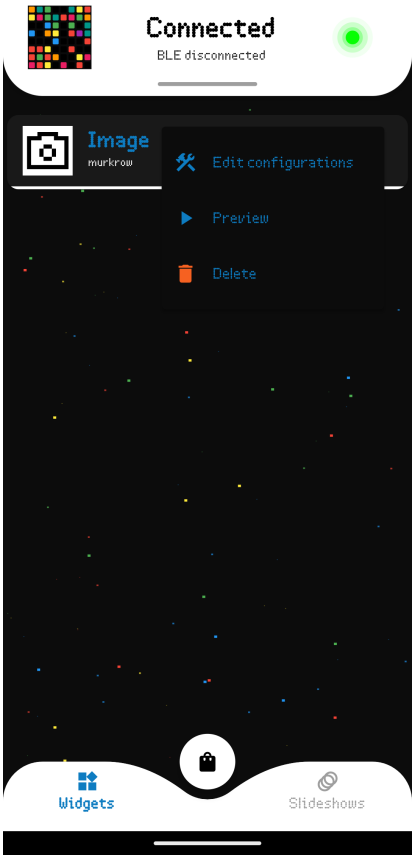


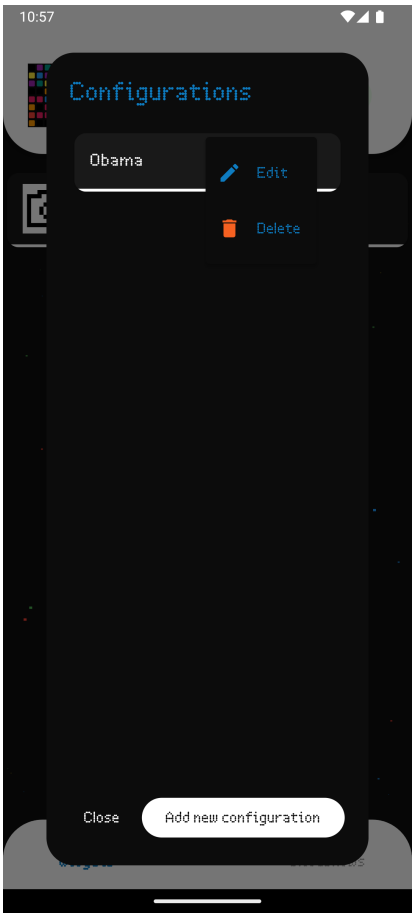


Once clicked on a widget, the user can see the widget details, by clicking on the installation button, the Raspberry Pi will proceed to git clone the widget and configure it for us. Once the widget has been installed it will be ready in the widgets tab.

2.2 Installed widgets

This is the first screen displayed when the app is launched. From here, the user can see, configure and manage the widgets installed on the Raspberry Pi.



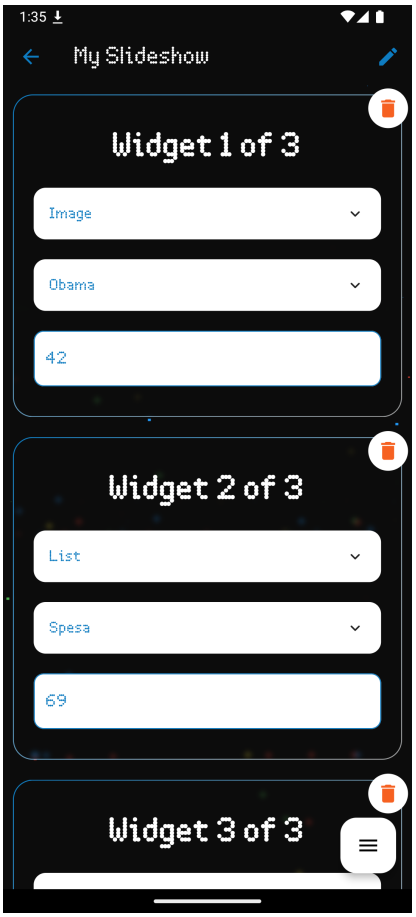


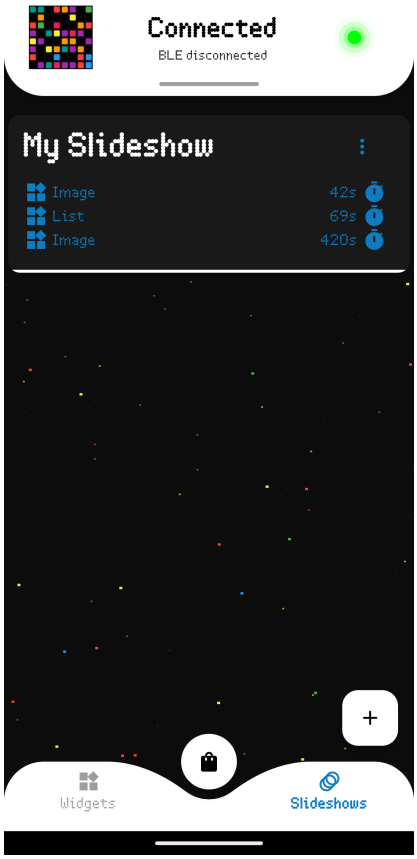
We can define 3 types of widgets:

- **Static**: Widgets that don't require any configurations
  - A clock
  - An inspirational quote
  - News headlines
  - Stock prices
- **Configurable**: Widgets that require some input from the user before being displayed
  - Weather forecast
  - Todo list
  - Image to pixel art
- **Interactive**: Widgets that require live interaction with the user (coming soon)
  - Games
  - Painter

2.3 Slideshows

The slideshow feature allows the user to create a sequence of widgets that will be displayed in a loop. The user can define the duration of each widget in seconds.





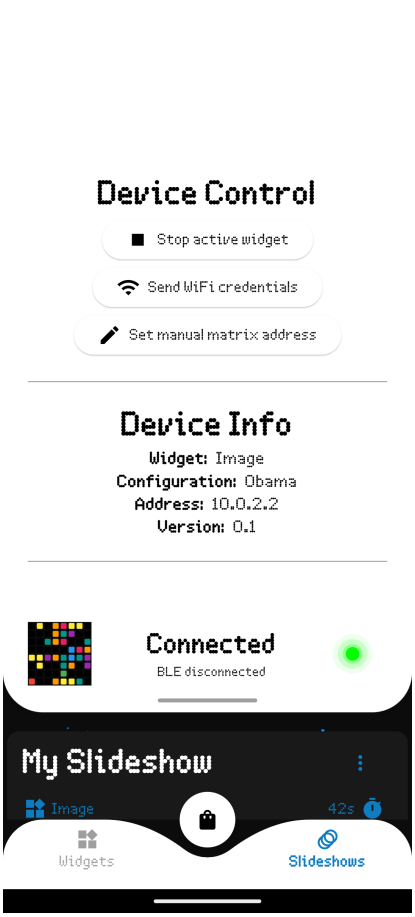
Slideshows items are easily draggable and can be reordered at will. It is always possible to add, remove or edit the widgets in the slideshow.

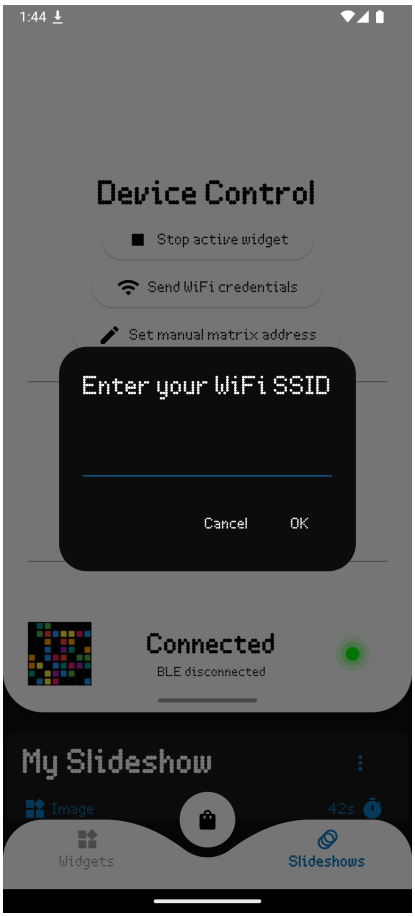
2.4 Device status

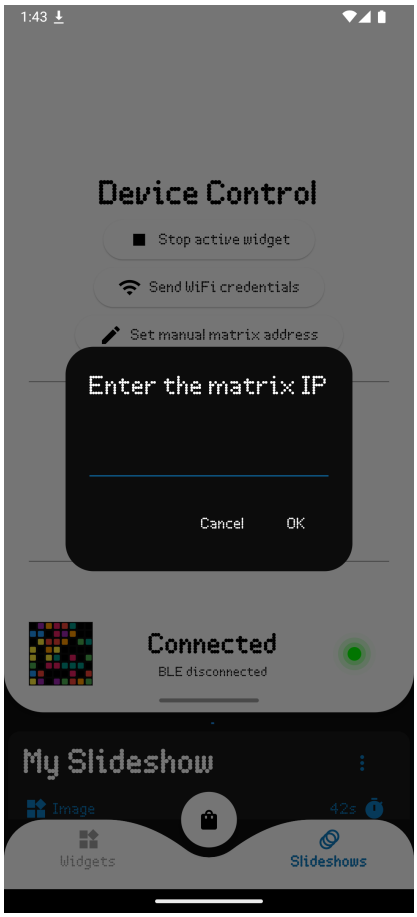
When in the home screen, the status of the connection to the Raspberry Pi is always displayed. In particular, we consider the app `connected` to the device when a COAP connection can be established, a warning will be displayed if BLE is not connected.

All the major features of the app are available only when the app is connected to the Raspberry Pi via COAP while BLE is only used to find the matrix on the local network (discovery) or to send network credentials in case the matrix is not connected to the network yet.

By pulling down the sliding panel we can obtain more information about the device, such as the IP address, the firmware version, and the current widget/configuration being displayed.







3. Shared library

3.1 Introduction

Since this project is composed of multiple applications, I preferred to create a shared library to store common code that can be used across different projects.

This library is written in Dart and mostly contains classes and functions that are crucial for the communication with the matrix or common UI components to share between the IDE and the mobile app. The library is divided into the following folders:

3.2 Project structure

- **core**:
  - **configurations**:
    - `app_color_scheme.dart` : Contains the color scheme used in the app. This is shared so that both the IDE and the mobile app can use the same colors.
    - `coap_config.dart` : Contains the configuration for the CoAP client.
    - `configs.dart` : Contains the configuration for the app.
  - **exceptions**: Contains custom exceptions that can be thrown by the app and caught by the global error handler in order to dispatch the error to the UI.
  - **models**: Shared models needed throughout the app.
  - **networking**:
    - `ble_service.dart` : Contains the BLE client that is used to communicate with the matrix.
    - `coap_service.dart` : Contains the CoAP client that is used to communicate with the matrix.
    - `rest_service.dart` : Contains the REST client that is used to communicate with the API.
- **utils**:
  - **common**: shared dart widgets used in IDE and APP
    - matrices
      - `led_matrix.dart` : The main matrix widget. This can be extended with multiple animations and effects.
      - `loading_matrix.dart` : Random pixels turned off and on to simulate loading.
      - `no_data_matrix.dart` : A widget that displays animated sad face with a message when there is no data to display.
- **features**:
  - **config_generator**: the module responsible for generating the configuration file for a widget with the `dynamic_form`
  - **matrix_control**: the module responsible for controlling the matrix and checking the connection status
  - **mosaico_loading**: wrapper around pages to handle loading states
  - **mosaico_slideshows**: the module responsible for programming slideshows
  - **mosaico_widgets**: the module responsible for installing and managing widgets

4. Configuration generator

One of the most interesting features of the shared library is the configuration generator.

This module can parse arbitrary .json files (also .yml in future) and convert them into a flutter form, which can be used to request data from the user to create a widget configuration. This choice was crucial to allow developers to create new widgets without having to worry about the UI, and to have consistency in the configuration of widgets from different developers.

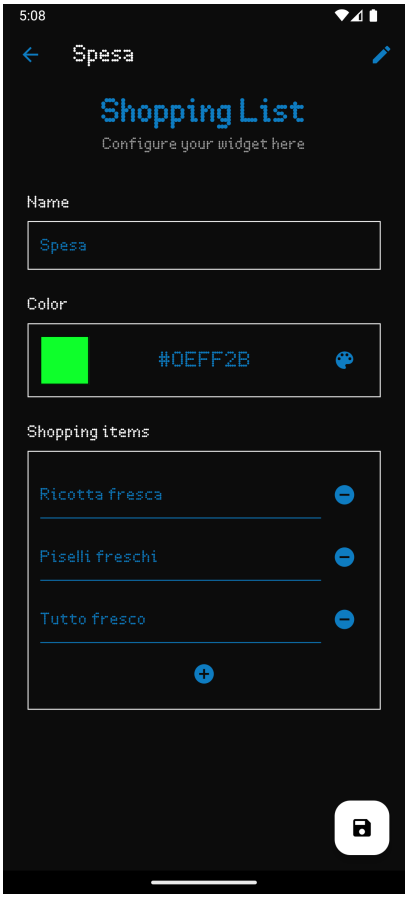
4.1 Examples

A simple example of a configuration file to create a configurations to upload an image to the matrix could be:

```
{
 "form": {
  "title": "Image",
  "description": "Simply choose an image to display in pixel art on the matrix!",
  "fields": [
   {
    "image": {
     "type": "image",
     "label": "Image",
     "required": true
    }
   }
  ]
 }
}
```

A simple example of a configuration file for a shopping list widget could be:

```
{
 "form": {
  "title": "Shopping List",
  "description": "Configure your widget here",
  "fields": [
   {
    "name": {
     "type": "string",
     "label": "Name",
     "required": true,
     "placeholder": "Enter the name of this list"
    },
    "color": {
     "type": "color",
     "label": "Color",
     "required": true,
     "placeholder": "Choose the color of the title"
    },
    "items": {
     "type": "string[]",
     "label": "Shopping items",
     "required": true,
     "placeholder": "Insert your items here"
    }
   }
  ]
 }
}
```



4.2 Result of the configuration

The data collected from the user is then saved in a .json file, which is then parsed by C++ when a specific widget with that configuration is loaded. Getting data from the configuration is very easy and can be done in python with the following code:

```
from mosaico import widget, config # config is the object holding all data from the configuration

text = widget.createText()
text.setText(config["name"]) # Get the name of the list from the configuration
text.setHexColor(config["color"]) # Get the color of the heading from the configuration
text.translate(2,2)
text.setFontHeight(10)

# Create items
items = []
for i in range(0, len(config["items"])): # We can iterate in the array provided by the user
    items.append(widget.createText())
    items[i].setFontHeight(6)
    items[i].setText(config["items"][i])
    items[i].translate(2, 6+ 7 * (i + 1))

def loop():
    pass
```

4.2.1 What about the assets?

User uploaded assets like images are saved in the .tar.gz archive in the assets folder, a helper function is available to retrieve the path of the asset in the archive.

```
from mosaico import widget, config

# Create image
img = widget.createImage(widget.configAsset("image")) # The name of the asset

def loop():
    pass
```

4.3 Implementation

4.3.1 Flutter

The configuration generator (and its flutter component, `dynamic_form`) are made up with extensibility in mind, so that's easy to add new field types. The `dynamic_form_state_builder` is responsible to parse the config-form.json file and create the form state. The implementation is easy and uses inheritance to easily manage the similarities between the different field types.

```
class DynamicFormStateBuilder {

 /// The form that is being built
 DynamicFormState _formModel = DynamicFormState();


 /// Adds the common attributes to a generic mosaico component
 void _addComponentAttributes(MosaicoField component,
    Map<String, dynamic> attributes, String fieldName) {
  component.getState().setName(fieldName);
  component.getState().setLabel(attributes['label'] ?? fieldName);
  component.getState().setPlaceholder(attributes['placeholder'] ?? '');
  component.getState().setRequired(attributes['required']);
  component.getState().setOldConfigPath(_formModel.getOldConfigDirPath());
 }

 DynamicFormStateBuilder(Map<String, dynamic> configForm) {

  // Get the main form
  var form = configForm['form'];

  // Set title and description
  _formModel.setTitle(form['title']);
  _formModel.setDescription(form['description']);

  // Get fields
  var fields = form['fields'];

  // Cycle through all fields
  for (var field in fields) {
   // Get field key (name of the field)
   for (var fieldName in field.keys) {
    // Retrieve field attributes
    var attributes = field[fieldName];

    // Add the final field to the form based on its type
    MosaicoField mosaicoField;
    switch (attributes['type']) {
     case 'string':
      mosaicoField = MosaicoStringField();
      break;
     case 'string[]':
      mosaicoField = MosaicoStringListField();
      break;
     case 'image':
      mosaicoField = MosaicoImageField();
```

```
          break;
        case 'color':
          mosaicoField = MosaicoColorField();
          break;
        case 'checkbox':
          throw Exception('Checkbox field not implemented yet');
          break;
        case 'animation':
          throw Exception('Animation field not implemented yet');
          break;
        default:
          throw Exception('Unknown field type: ${attributes['type']}');
      }

      // Add common attributes
      _addComponentAttributes(mosaicoField, attributes, fieldName);

      // Add the field to the form
      _formModel.addField(mosaicoField);
    }
   }
  }


  DynamicFormState buildFormModel() {
   return _formModel;
  }

}
```

Getting values from the fields is easy too, all the fields use a common state class that needs to implement the `MosaicoFieldState` interface.

```
abstract class MosaicoFieldState extends ChangeNotifier {

  /*
  * Name
  */
  late String _name = "";
  void setName(String name) {
    _name = name;
  }
  String getName() {
    return _name;
  }

  /*
  * Label
  */
  late String _label;
  void setLabel(String label) {
    _label = label;
  }
  String getLabel() {
    return _label;
```

```
  }

  /*
  * Placeholder
  */
  String? _placeholder;
  void setPlaceholder(String placeholder) {
    _placeholder = placeholder;
  }
  String? getPlaceholder() {
    return _placeholder;
  }

  /*
  * Required
  */
  bool _required = false;
  void setRequired(bool required) {
    _required = required;
  }
  bool isRequired() {
    return _required;
  }

  /*
  * Edit mode
  */
  String? _oldConfigPath;
  bool get isEditMode => _oldConfigPath != null && _oldConfigPath!.isNotEmpty;
  void setOldConfigPath(String oldConfigPath) {
    _oldConfigPath = oldConfigPath;
  }
  String? getOldConfigPath() {
    return _oldConfigPath;
  }

  /*
  * Stuff to override
  */

  /// Called when form is submitted
  /// Return null if valid, error message if not
  String? validate();

  /// This method should return the field data to be available in the config json
  dynamic getData();

  /// Optional override to save an asset into the config
  dynamic getAsset();

  /// Provides the field with the old value previously saved with getData
  void init(dynamic oldValue);
}
```

This ensures that in a later stage, when the form is submitted, we can easily get the data from the fields and save it in a .json file. This is done by the `config_form_state` file that manages the form state and the form submission, and the `config_output` file that manages the final output of the form.

```
/*
* Final output
*/
 String buildConfigJson() {
   Map<String, dynamic> editJson = {};
   for (var field in _fields) {
     var fieldData = field.getState().getData();
     if (fieldData == null) continue;
     editJson[field.getState().getName()] = field.getState().getData();
   }
   return jsonEncode(editJson);
 }

 // Export the form to a ConfigOutput object
 Future<ConfigOutput> export() async {
   var output = ConfigOutput();
   await output.initialize();

   // Config name
   output.setConfigName(_configName);

   // Config json
   output.saveConfigJson(buildConfigJson());

   // Assets
   for (var field in _fields) {
     if (field.getState().getAsset() != null) {
       output.saveAsset(
           field.getState().getName(), field.getState().getAsset()!);
     }
   }

   return output;
 }
```

4.3.2 Files

At the end of the day, a configuration is just a .tar.gz archive, created locally on the device with the archive package. The `config_output` class is responsible for creating the archive and saving the json file.

```
class ConfigOutput {

  var logger = Logger(
    printer: PrettyPrinter(),
  );

  late final String _tempPath;
  late final String _dataOutputPath;
  late final String _configName;

  /// Initialize the _tempPath
  /// CALL THIS BEFORE SETTING ANYTHING!
  Future<void> initialize() async {

    // Create temp path
    Directory tempDir = await getTemporaryDirectory();
    _tempPath = tempDir.path + "/config_output";
    _dataOutputPath = _tempPath + "/data";

    // Create new directories
    Directory(_tempPath).createSync();
    Directory(_dataOutputPath).createSync();
  }

  /**
   * Configuration name
   */
  void setConfigName(String name) {
    _configName = name;
  }
  String getConfigName() {
    return _configName;
  }

  /**
   * Save files to disk in temp folder as they are provided
   */
  void saveConfigJson(String json) {
    File('$_dataOutputPath/config.json').writeAsStringSync(json);
  }

  void saveAsset(String assetName, String assetContent) {
    // Create assets directory if it doesn't exist
    Directory('$_dataOutputPath/assets').createSync();
    File('$_dataOutputPath/assets/$assetName').writeAsStringSync(assetContent);
  }

  /**
   * Export everything
```

```
   */

  /// Exports the created directory with config files to a .tar.gz archive
  /// Returns the path to the archive
  String exportToArchive() {

    // Define the source directory and the output file
    final sourceDir = Directory(_dataOutputPath);
    final outputFile = File('$_tempPath/config.tar.gz');

    // Create an archive
    final encoder = TarFileEncoder();
    encoder.open('$_tempPath/config.tar');

    // Add the contents of the directory to the archive
    encoder.addDirectory(sourceDir, includeDirName: false);

    // Close the archive to finalize the tar file
    encoder.close();

    // Read the tar file
    final tarFile = File('$_tempPath/config.tar');
    final tarBytes = tarFile.readAsBytesSync();

    // Compress the tar file using gzip
    final gzBytes = GZipEncoder().encode(tarBytes);

    // Write the compressed file to disk
    outputFile.writeAsBytesSync(gzBytes!);

    // Optionally delete the intermediate tar file
    tarFile.deleteSync();

    logger.d('Exported config to: ${outputFile.path}');
    return outputFile.path;
  }


}
```

5. Implementation details

5.1 Why Flutter?

This is the first time I've used Flutter, and I must say I'm impressed. The development process is smooth, hot-reload is a game-changer, the pub.dev ecosystem is rich, and the community is very active. I've always been able to find a package that does what I need, and when I couldn't, I was able to write my own without too much hassle.

I found it really easy to create a beautiful and responsive UI, the only steep learning curve was the state management, at first I used the standard `setState` of the `StatefulWidget` but then I switched to `Provider` and `ChangeNotifier` to keep the UI code totally separated from the business logic. Using the `Provider` package also allowed me to move the providers I needed in both the mobile app and the IDE to the shared library, so I could reuse them in both projects without duplicating code.

5.2 Networking

The app uses multiple network protocols to communicate with the Raspberry Pi and the App Store. Obviously, the Pi-0 is not a beast in terms of computation power, so I tried to keep the network traffic and resource usage as low as possible, also considering that most of the resources will be used to render the widgets on the LED matrix. A full-fledged REST API would have been overkill for this project, so after consulting with my professor, I discovered the COAP protocol, which is perfect for IoT devices. To save data on the raspberry pi, I decided to use a lightweight SQLite database.

5.2.1 Coap

COAP let me create a simple and efficient API without the bloated overhead of HTTP, and it's perfect for the Raspberry Pi, which has limited resources. The following endpoints are exposed by the Raspberry Pi:

- `/widgets/installed` : the installed widgets
  - **GET**: returns the list of installed widgets
  - **POST**: installs a new widget
  - **DELETE**: uninstalls a widget
- `/widgets/active` : the active widget
  - **GET**: returns the active widget
  - **DELETE**: stops the active widget
- `/widgets/configuration_form` : the form used as 'blueprint' to create a new widget configuration
  - **GET**: returns the form
- `/widgets/developed` : the widgets developed by the user (usually with the IDE)
  - **POST**: installs a new widget
- `/widget_configurations/{widget_id}` : the configurations of the widgets
  - **GET**: returns the list of configurations for a widget
  - **POST**: creates a new configuration
  - **DELETE**: deletes a configuration
- `/widget_configurations/package/{config_id}` : the package generated by a widget configuration, used to edit an existing configuration
  - **GET**: returns the package generated by a configuration
- `/slideshows` : the slideshows created by the user
  - **GET**: returns the list of slideshows
  - **POST**: creates a new slideshow
  - **DELETE**: deletes a slideshow
- `/slideshows/active` : the active slideshow
  - **POST**: starts a slideshow

Note: the COAP server is implemented in Python using the `aiocoap` library. The library didn't support templated urls (like `/widget_configurations/{widget_id}`) so I had to create a custom router to handle these type of requests.

5.2.2 BLE

Bluetooth Low Energy is an extremely power-efficient protocol that allows the mobile app to talk with the Raspberry Pi even when the matrix is not connected to the local network. The BLE server is implemented in Python using the `bless` library, it uses the service UUID `d34fdcd0-83dd-4abe-9c16-1230e89ad2f2` and it exposes the following characteristics:

- **9d0e35da-bc0f-473e-a32c-25d33eaae17a**: Discovery
  - READ: used to discover the matrix and returns `Hello World!` message
- **9d0e35da-bc0f-473e-a32c-25d33eaae17b**: Wi-Fi
  - WRITE: used to send the Wi-Fi credentials to the Raspberry Pi to connect to the local network
- **9d0e35da-bc0f-473e-a32c-25d33eaae17c**: IP
  - READ: used to get the IP address of the Raspberry Pi

I chose to use BLE for the following reasons:

- It's power-efficient
- It's supported by most devices
- It allows the app to communicate with the Raspberry Pi even when the matrix is not connected to the local network
- Latency generally is lower than Wi-Fi, this can be useful for live-interaction widgets in the future

I didn't use BLE to handle the main communication between the mobile app and the Raspberry Pi for the following reasons:

- BLE throughput is much narrower than Wi-Fi
  - This could be a problem when sending files back and forth like the widget configuration packages
- BLE has a limited range compared to Wi-Fi
- BLE GATT server is much more complex to set up rather than a COAP server

5.2.3 REST

On the other hand, on a mobile device we don't have the same constraints as on the Raspberry Pi, so I decided to expose a simple REST API to handle the App Store. The API is written in PHP using the Laravel framework, for now is very minimal, and it exposes the following endpoints:

- `/widgets` : the widgets available in the store
  - **GET**: returns the list of all the widgets
    - This can be filtered by various parameters like the category, the type, the name, etc. using query parameters like `?sort=name:asc`
- `/widgets/{widget_id}` : the details of a widget
  - **GET**: returns the details of a widget, more in detail rather than the list of widgets

6. Credits

6.1 Developers

This project was developed by the following people:

- murkrow: Project lead, developer

6.2 Open source community ❤

This project wouldn't have been possible without the awesome open source community, here are some of the libraries and resources used in this project:

- hzeller: Huge shoutout to Henner Zeller for the awesome C++ library to control the LED matrix
- Adafruit: LED matrix bonnet and guide to set up the pi
- Nlohmann JSON: Modern JSON library for C++
- pybind11: Amazing seamless operability between C++ and Python
- aiocoap: Python library to create COAP server
- bless: Python library used to set up a simple GATT server with BLE
- GitPython: Python library to interact with git repositories
- PixelArtIcons: Used in the app store with some widgets
- Laravel: PHP framework used to create the App Store API
- Flutter: Framework used to create the mobile app

6.3 Tools

- Raspberry Pi: Used to control the LED matrix
- Raspberry Pi cross-compiler: Used to compile the C++ code for the Raspberry Pi without needing to wait for hours
- Android Studio: Used to develop the Android app
- PHPStorm: Used to develop the App Store API
- CLion: Used to develop the LED matrix controller
- PyCharm: Used to develop the networking module
- Docker: Simplified the development and deployment process by containerizing the app store API
- ChatGPT: Helped me a lot to learn flutter and python
- SQLite: Used to store the data in the networking module