

18 Implementation Admin Panel

The Admin Panel is a centralized administrative dashboard to efficiently manage all addresses within a single environment. It is used to plan for future "Sternsinger" events, ensuring that every address that needs to be visited is covered and that all addresses are efficiently distributed among participating groups. Additionally, it enables the assignment of areas containing addresses a group needs to visit. This zoning feature ensures that each group is responsible for visiting only the addresses within their assigned area.

With the tool, administrators can perform CRUD (Create, Read, Update, Delete) operations on addresses, streets, and areas. These features make it easy to quickly address issues and make changes to the areas that participants need to visit. For example, if a new street is added to the neighborhood, the administrator can update the system to include this street and assign it to the appropriate area. Similarly, if a group drops out, the administrator can quickly reassign the addresses that have not yet been visited to other groups. This ensures that the data remains up-to-date and allows for quick reactions to special cases, helping with the planning and execution of "Sternsinger" events.

This chapter will outline the implementation of the Admin Panel and describe the different components, functionalities, and widgets of this tool. Additionally, it will provide information on how to use it.

18.1 Navigation

To navigate between pages, a sidebar on the left is used, which can be toggled with a button in the top-left corner of the screen. It shows a list of all pages, allowing users to switch between them with a click.

The navigation is implemented in the file `AdminNavigation`. It contains a list of pages with titles. These titles are displayed at the top of the screen above the corresponding page. To keep track of the currently selected page, an internal state (`indexState`) is used. Whenever a page is selected in the sidebar, the `indexState` is updated, and the corresponding page is displayed. This widget is the main component. It makes sure that all pages are properly displayed.

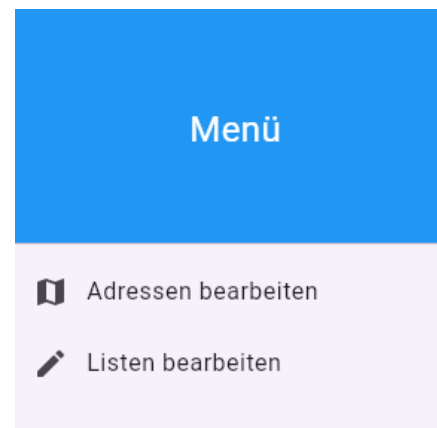


Abb. 1: Navigation in Admin-Panel

18.2 AddressPage

The first page, the `AddressPage`, displays an interface to create, update, and delete address data. It uses a form with various input fields (e.g., street, house number, coordinates, special features) and a map or database view to visualize and select addresses. The page is divided into two parts:

- On the right side, all addresses are shown either in the `AdminMapComponent` (18.4.1) or the `DatabaseViewComponent` (18.4.2). These two components display the same addresses but in different ways, to give the administrator the choice of how they want to view the addresses.
- On the left side of the page are `InputFields` (18.6.1), which are used to enter new information about a new address or edit an existing one.

Overlaying the `AdminMapComponent`, there are:

- A field to filter the addresses displayed.
- A button with a dropdown menu to select and edit a street.
- A switch to toggle between the `AdminMapComponent` and the `DatabaseViewComponent`.
- An information box in the bottom left corner to display `Notifications` (18.2.5.1) about the completed operations.
- A field in the bottom right corner to display the coordinates of the mouse pointer on the map.

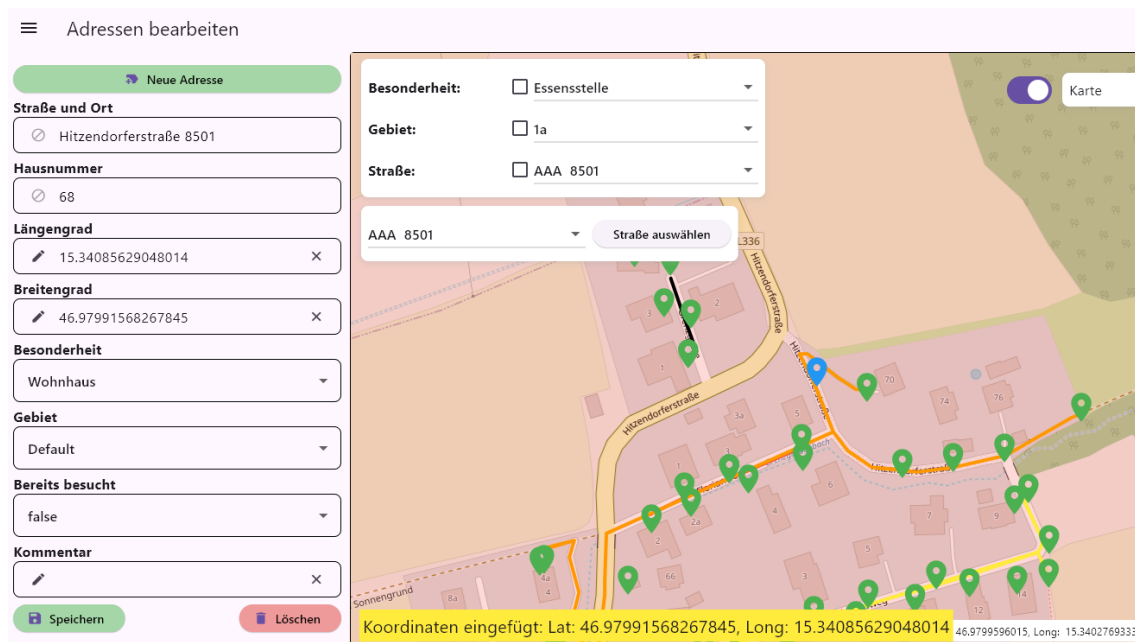


Abb. 2: AddressPage

18.2.1 Add Address

To add a new address, the button "Neue Adresse" is pressed. A single click on the map does the same thing, however it automatically fills in the coordinates of the location where the mouse was clicked. This triggers the `onClickNewAddress` method, which performs several key operations:

- All `InputFields` are cleared.
- The boolean variable `isNewAddress` is set to `true`, indicating that a new address is being created.

The cleared fields can then be filled with the new information. When the "Speichern" button is pressed, the `saveAddress` method is called. This method performs several validation checks which can be seen in 18.2.4. If the address is valid, the `AdminAddressProvider` is called to add it to the database. If the operation was successful, a `Notification` is displayed and the newly added address appears on the map.

18.2.2 Edit Address

Existing addresses can be edited by selecting an address in either the `AdminMapComponent` or the `DatabaseViewComponent`. The selection fills the `InputFields` with the information of the selected address, and in this case, the boolean variable `isNewAddress` is set to `false` to indicate that an existing address is being updated.

The admin can then edit the information and press the "Speichern" button. This triggers the `saveAddress` method, which performs the same validation checks as when adding a new address. The `AdminAddressProvider` updates the selected addresses in the database, and the `Notification` is shown.

18.2.3 Delete Address

After selecting an address, the admin can press the "Löschen" button to delete it. This triggers the `deleteAddress` method, which calls the `AdminAddressProvider` to delete the selected addresses and displays the `Notification`.

The method also triggers the `showDeleteDialog` method, which displays a `AlertDialog` to confirm the action and prevent accidental deletions.

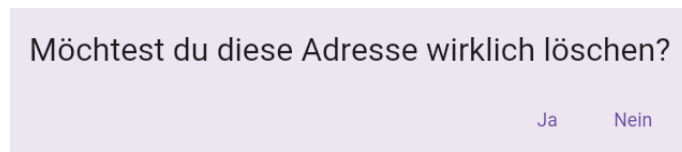


Abb. 3: Dialog to confirm deletion

18.2.4 Validation

Validation is the process of checking that data meets specific criteria before it is accepted and added. It is crucial to ensure that the data entered is correct, consistent, and meets the required standards to maintain data integrity and reliability. (Contributors to Wikimedia projects, 2025)

18.2.4.1 isDuplicateAddress

To determine whether an edited or newly added address already exists, All addresses are compared with the new address. It is called in the `saveAddress` method. If it already exists, the method returns `true`, otherwise `false`. A duplicate address is identified by the following criteria:

- street name
- postal code
- house number

```
bool isDuplicateAddress(List<Address> existingAddresses, Address
newAddress) {
    return existingAddresses.any((existing) =>
        existing.street.name == newAddress.street.name &&
        existing.street.postalCode == newAddress.street.postalCode &&
        existing.houseNumber == newAddress.houseNumber
    );
}
```

Quellcode 1: isDuplicateAddress method

18.2.4.2 InputField filled Validation

To make sure that all `InputFields` are filled, the `validateAddressFields` is called in the `saveAddress` (18.2.1) method. This method needs an `Address`. So first an new `Address` is made and passed to `validateAddressFields`. It checks if all fields are filled and returns `true` if they are, otherwise `false`.

```
bool validateAddressFields(Address address) {
    if (address.street.name.isEmpty) {
        showNotification("Strasse fehlt", () => showAddAddressNotification =
            true);
    } else if (address.houseNumber.isEmpty) {
        showNotification("Hausnummer fehlt", () => showAddAddressNotification
            = true);
    } else if (address.specialFeature.text.isEmpty) {
        showNotification("Besonderheit fehlt", () =>
            showAddAddressNotification = true);
    }
}
```

```

    } else if (address.area.desc.isEmpty) {
        showNotification("Gebiet fehlt", () => showAddAddressNotification =
            true);
    } else if (address.latitude == 0.0 || address.longitude == 0.0) {
        showNotification("Koordinaten fehlen", () =>
            showAddAddressNotification = true);
    } else {return true;}
    return false;
}

```

Quellcode 2: InputFormatter in Inputfield

18.2.4.3 InputField Coordinates Validation

The `InputField` validates **latitude** and **longitude** inputs to ensure their correctness. Validation is applied only when the `isNumberInput` parameter is set to true. If that is the case, then the `inputFormatter` is passed to the `textfield` to validate the input. This `inputFormatter` guarantees that only valid inputs are accepted, preventing incorrect entries. These are the three validators used in the `InputField`:

- The `FilteringTextInputFormatter` allows only numbers and dots with a regular expression.
- The `TextInputFormatter` checks if the input contains more than one dot and checks that there are no more than three digits before the decimal point.

```

inputFormatters: widget.isNumberInput == true
? <TextInputFormatter>[
    FilteringTextInputFormatter.allow(RegExp('[0-9.]')),
    TextInputFormatter.withFunction((oldValue, newValue) {
        String text = newValue.text;
        if (text.split('.').length - 1 > 1 || (text.isNotEmpty &&
            text.split('.')[0].length > 3)) {
            return oldValue;
        }
        return newValue;
    })
]
: null,

```

Quellcode 3: InputFormatter in Inputfield

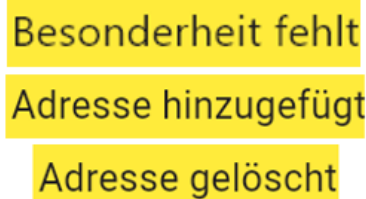
18.2.5 Additional Functionalities

This section highlights various functionalities implemented to improve the overall usability of the application. These additions are designed to support administrative tasks and ensure a seamless user experience.

18.2.5.1 Notification

To inform the administrator about the success or failure of an operation, a `Notification` is displayed on the bottom left, overlaying the `AdminMapComponent`. This notification appears when:

- An address is added, edited, or deleted.
- Validation fails.
- Coordinates are selected on the `AdminMapComponent` (18.4.1.3).



Besonderheit fehlt

Adresse hinzugefügt

Adresse gelöscht

Abb. 4: Notification examples

To display this notification, the `showNotification` method is called. This method sets the `notificationVisible` variable to `true` and starts a `Timer` to turn it set it back to `false` three seconds later, so it goes away in a short time. This method accepts a message as a parameter, which is saved in the `notificationText` variable.

```
void showNotification(String message) {
    setState(() {
        notificationText = message;
        notificationVisible = true;
    });
    Timer(Duration(seconds: 5), () {
        setState(() {
            notificationVisible = false;
        });
    });
}
```

Quellcode 4: showNotification method

When the `notificationVisible` variable is set to `true`, the UI-component which shows the notification is rendered.

```
Positioned(
  left: 10,
  bottom: 10,
  child: (notificationVisible)
    ? Container(
      padding: EdgeInsets.all(4),
      color: Colors.yellow,
      child: Text(
        notificationText,
        style: TextStyle(fontSize: 20),
      ),
    )
    : Container(),
),
```

Quellcode 5: Notification in AddressPage

18.2.6 Edit multiple addresses

To make it easier to edit multiple addresses at once, the CTRL-Button on the keyboard is listened to. When the CTRL-Button is pressed, the `isCtrlPressed` variable is set to `true`. This variable is passed to the `DatabaseViewComponent` and the `AdminMapComponent`, to inform them that multiple addresses want to be selected. The components can use the, as a parameter given, `markerSelected` function to set the selected addresses in the `AddressPage`. The `saveAddress` method is called to save multiple addresses.

To listen to the CTRL-Button, the predefined `RawKeyboardListener` is used. This widget sets the `isCtrlPressed` variable to `true` when the CTRL-Button is pressed and to `false` when it is released. To make sure that repeatedly pressing the CTRL-Button, which happens if the button is pressed and held, does not interfere with the `isCtrlPressed` variable, the condition `event.repeat == false` is used.

```
RawKeyboardListener(
  autofocus: true,
  focusNode: FocusNode(),
  onKey: (event) {
    if ((event.isKeyPressed(LogicalKeyboardKey.controlLeft) ||
      event.isKeyPressed(LogicalKeyboardKey.control) ||
      event.isKeyPressed(LogicalKeyboardKey.controlRight)) &&
      event.repeat == false) {
      setState(() {
        isCtrlPressed = true;
      });
    }
  },
)
```

```

    });
  } else if (event is RawKeyUpEvent && event.logicalKey ==
    LogicalKeyboardKey.controlLeft) {
    setState(() {
      isCtrlPressed = false;
    });
  }
},
),

```

Quellcode 6: RawKeyboardListener in AddressPage

The `markerSelected` method also updates the `InputField` for the house number by listing the house numbers of the selected addresses in the controller.

```

controllers["houseNumber"]?.text = selectedAddresses
  .map((address) => address.houseNumber)
  .toList()
  .join(', ');

```

Quellcode 7: Listed house numbers for multiple selected addresses

The `InputField` looks like this:

Hausnummer

Abb. 5: House numbers of multiple selected addresses

The selected addresses are saved in the `selectedAddresses` variable in the `AddressPage`. If multiple addresses are selected, certain `InputFields` such as house numbers, coordinates, or comments will be disabled because changing them for all selected addresses would not make sense. This can be achieved by setting the `editable` parameter of these `InputFields` to `selectedAddresses.length <= 1`, so that they are only editable when a single address is selected.

18.2.7 Edit all Addresses from a Street

All addresses from a street can be edited at once. This is almost the same as editing multiple addresses (18.2.6), but instead of selecting the addresses by clicking on them, a street is selected. When a street is selected, all its addresses are selected. A street can be selected in two ways:

18.2.7.1 Select Street via AdminMapComponent

Every click on the `AdminMapComponent` is checked if it is near a street. Because there is no predefined method to check if a point is on a street, the `isPointNearPolyline` method was implemented. If the

point is near a street, the method returns `true` otherwise `false`. More information about this method can be found in the `AdminMapComponent` section 18.4.1.

18.2.7.2 Select Street via Button

This was implemented because we encountered a problem where it was not possible to click on a street when the Admin Panel was deployed. This problem only occurred on some devices. To ensure that the Admin Panel is usable on all devices, a dropdown button to choose a street and a button to select it were implemented.



Abb. 6: Button to select a street

18.2.8 Edit Odd / Even Streets

One requirement was that addresses from one street could be automatically added to two areas based on whether the house number is even or odd. This is because it is common for all addresses with even house numbers to be on one side of the street and those with odd house numbers on the other side. This way, it is easier to assign the street sides to different areas, so that "Sternsinger" participants don't have to cross the street so often.

To make this possible, the administrator has to select a street in the `AdminMapComponent` (18.4.1.1), then a blue button beneath the `InputFields` appears.

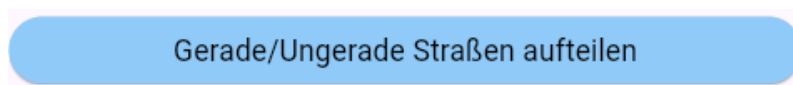


Abb. 7: Button to split street

After pressing this button, a dialog appears, where the administrator can select the areas for the addresses with even and odd house numbers.

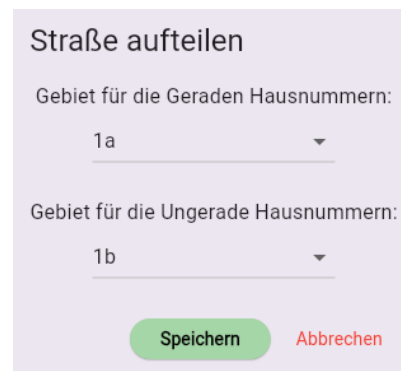


Abb. 8: Dialog to split street

The "Speichern" button triggers the `AdminAddressProvider` and shows a `Notification` (18.2.5.1) if the operation was successful or not. After that, the dialog is closed.

```
int resultCode = await
    addressProvider.putAddressesOddEven(selectedAddresses.first.street.name,
        selectedAddresses.first.street.postalCode, selectedAreaEven,
        selectedAreaOdd);
if (resultCode == 200) {
    showNotification("Adressen gespeichert", () =>
        showAddressSavedNotification = true);
} else {
    showNotification("Fehler beim Speichern", () => showAddAddressNotification
        = true);
}
if (!context.mounted) return;
Navigator.of(context).pop();
```

Quellcode 8: onPressed save button in splitStreetDialog

18.2.9 Filter

With the Filter field, the administrator can filter the addresses displayed. It contains three dropdown menus to set the filter criteria, with one checkbox for each to toggle them. These filters can be combined as desired.

Besonderheit:	<input type="checkbox"/> Besuch nicht gewünscht	▼
Gebiet:	<input type="checkbox"/> 1a	▼
Straße:	<input checked="" type="checkbox"/> Amselgasse 8501	▼

Abb. 9: Filter field in AddressPage

The filter is passed and applied to the `AdminMapComponent` and the `DatabaseViewComponent`. The criteria and their enabled/disabled state are managed by the following variables in the `AddressPage` class.

```
bool specialFeatureFilter = false;
bool areaFilter = false;
bool streetFilter = false;

String selectedStreetFilter = "";
String selectedSpecialFeatureFilter = "";
String selectedAreaFilter = "";
```

Quellcode 9: Filter variables in AddressPage

This is an example of how a `FilterRow` is defined in the `AddressPage` class (18.6.2):

```
FilterRow(
    label: "Besonderheit:",
    tooltipMessage: "Besonderheitsfilter aktivieren/deaktivieren",
    filterValue: specialFeatureFilter,
    onFilterChanged: (bool? newValue) {
```

```

        setState(() => specialFeatureFilter = newValue ?? false);
    },
    selectedValue: selectedSpecialFeatureFilter,
    items: specialFeatureTextList,
    onDropDownChanged: (String? newValue) {
        setState(() => selectedSpecialFeatureFilter = newValue ?? "");
    },
),

```

Quellcode 10: FilterRow in AddressPage

18.3 ListEditPage

The `ListEditPage` is used to manage **streets**, **special features**, and **areas**. It allows the administrator to add, edit, and delete these entities. The page is divided into two parts. On the right side, all entities are displayed in a table and can be selected. On the left, there is a dropdown menu for selecting between the three options. The information of the selected item is shown in `InputFields` on the this side, where the information can be edited, saved or deleted using the "Speichern" or the "Löschen" button.

Abb. 10: ListEditPage

18.3.1 QR-Code Visualization for Areas

To make it easier for users of the user application to get information about their area, a QR-Code is generated for every area, which can be scanned to get all information. The QR-Code contains the name of the area. The currently selected area is saved in the `selectedItem` variable.



Abb. 11: QR-Code for area

To display the QR-Code, the `QrImageView` Widget is used.

```
QrImageView (
  data: selectedItem,
  size: 300,
  padding: const EdgeInsets.all(16.0),
  gapless: false,
),
```

Quellcode 11: QrCode Generation in ListEditPage

18.3.2 QR-Code-PDF Download

A PDF with all QR-Codes for all areas can be downloaded, making it easier to distribute the areas to the users. The PDF is generated with the `savePDF` method of the `PDFSaver` class.

18.4 Components

18.4.1 AdminMapComponent

18.4.1.1 Select Street

18.4.1.2 onClickNewAddress

18.4.1.3 Select Coordinates on Map

18.4.2 DatabaseViewComponent

To display the addresses in a table, this component is used. As parameters it takes the `selectedAddresses`, the filter variables as well as the `isCtrlPressed` variable from the `AddressPage`. The

`selectedAddresses` are displayed in the table and can be selected by clicking on them.

On the top are two fields

18.4.3 PDFSaver

The `PdfSaver` class provides a static method to save a PDF file from a byte array. The method `savePdf` takes a `Uint8List` of bytes and a `String` representing the file name as parameters. Depending on the platform, it saves the PDF file accordingly.

```
static Future<void> savePdf(Uint8List bytes, String fileName) async {
  if (kIsWeb) {
    final blob = html.Blob([bytes], 'application/pdf');
    final url = html.Url.createObjectUrlFromBlob(blob);
    final anchor = html.AnchorElement(href: url)
      ..target = 'blank'
      ..download = fileName
      ..click();
    html.Url.revokeObjectUrl(url);
  } else {
    await FileSaver.instance.saveFile(
      name: fileName,
      bytes: bytes,
      ext: 'pdf',
      mimeType:MimeType.pdf,
    );
  }
}
```

Quellcode 12: savePdf method in PDFSaver

18.4.4 AdminAddressProvider

This class serves as a bridge between the Admin Panel and the backend. It is responsible for all CRUD operations on addresses, streets, special features, and areas. It is used in the `AddressPage` and the `ListEditPage`. The `AdminAddressProvider` includes the `ChangeNotifier` mixin. A mixin is a way to reuse code across multiple classes, without using inheritance like in Java. (“Mixins”, 2025) The `ChangeNotifier` mixin is used to notify the UI when the data changes. (“ChangeNotifier class - foundation library - Dart API”, 2025) This is done by calling the `notifyListeners` method.

Here is a typical method in the `AdminAddressProvider` class. It contains these functionalities:

- `async` : enables non-blocking operations and ensures that the UI remains responsive while waiting for tasks like network requests to complete.
- `await http.get` : sends a GET request to the server to fetch all streets and waits for the response.
- `jsonDecode` : processes the JSON response from the server.
- `utf8.decode` : transforms the UTF-8 encoded response body into readable text.
- `map` : converts the decoded JSON to a list of `Street` objects.
- `sort` : sorts the list alphabetically.
- `notifyListeners` : notifies the listeners that the data has changed.
- `catch` : catches any errors that occur during the operation.

```
Future<void> fetchStreets() async {  
  try {  
    final response = await  
      http.get(Uri.parse('$serverURL/$baseUrl/admin/getAllStreets'));  
    if (response.statusCode == 200) {  
      final List<dynamic> decodedJSON =  
        jsonDecode(utf8.decode(response.bodyBytes));  
      streets = decodedJSON.map((json) => Street.fromJson(json)).toList();  
      streets.sort((a, b) => a.name.compareTo(b.name));  
      notifyListeners();  
    } else {  
      throw Exception('Failed to load streets');  
    }  
  } catch (e) {  
    print('Error fetching streets: $e');  
  }  
}
```

Quellcode 13: typical method in AdminAddressProvider