GitHub Username: adrianrejas

Dashboard of Things

Description

Nowadays we have available a lot of cloud solutions oriented to IoT networks, such as Microsoft Azure IoT Hub, Google IoT core, IBM Watson IoT platform or Samsung Artik platform. These cloud platforms and many more provide us a bridge between IoT sensors and actuators (sensors like thermometers, movement detectors or heart rate trackers and actuators like smart plugs, thermostats or lights) and any device connected to the Internet. Most of these platforms relies on standard protocols, such as HTTP and MQTT.

The problem is the lack of a general frontend for accessing from there to different IoT backends. There are many IoT projects and installations which require just a simple frontend which can be configured for adapting to backend, but there are not many applications, no matter the platform, for that.

With Dashboard of Things, you'll be able to convert your phone or tablet in a dashboard where you'll be able to control your IoT devices remotely, receiving real-time and historical information from IoT sensors and being able to activate and configure your IoT actuators, being able to have IoT devices from different sources being monitored and controlled from a single device.

Intended User

This application targets two types of users:

- Users interested in monitoring and controlling IoT devices from their Android phone or tablet, no matter the IoT devices wanted to be managed use the same backend or different ones. They will be able to get info from the sensors they want at real time in a variety of ways.
- Users interested in transforming an old Android phone or tablet into a 24/7 dashboard for the monitoring and controlling of IoT devices. In these cases the dashboard will be shown at every moment, showing real time or historical info about IoT sensors and actuators.

Features

These are the main features of the app:

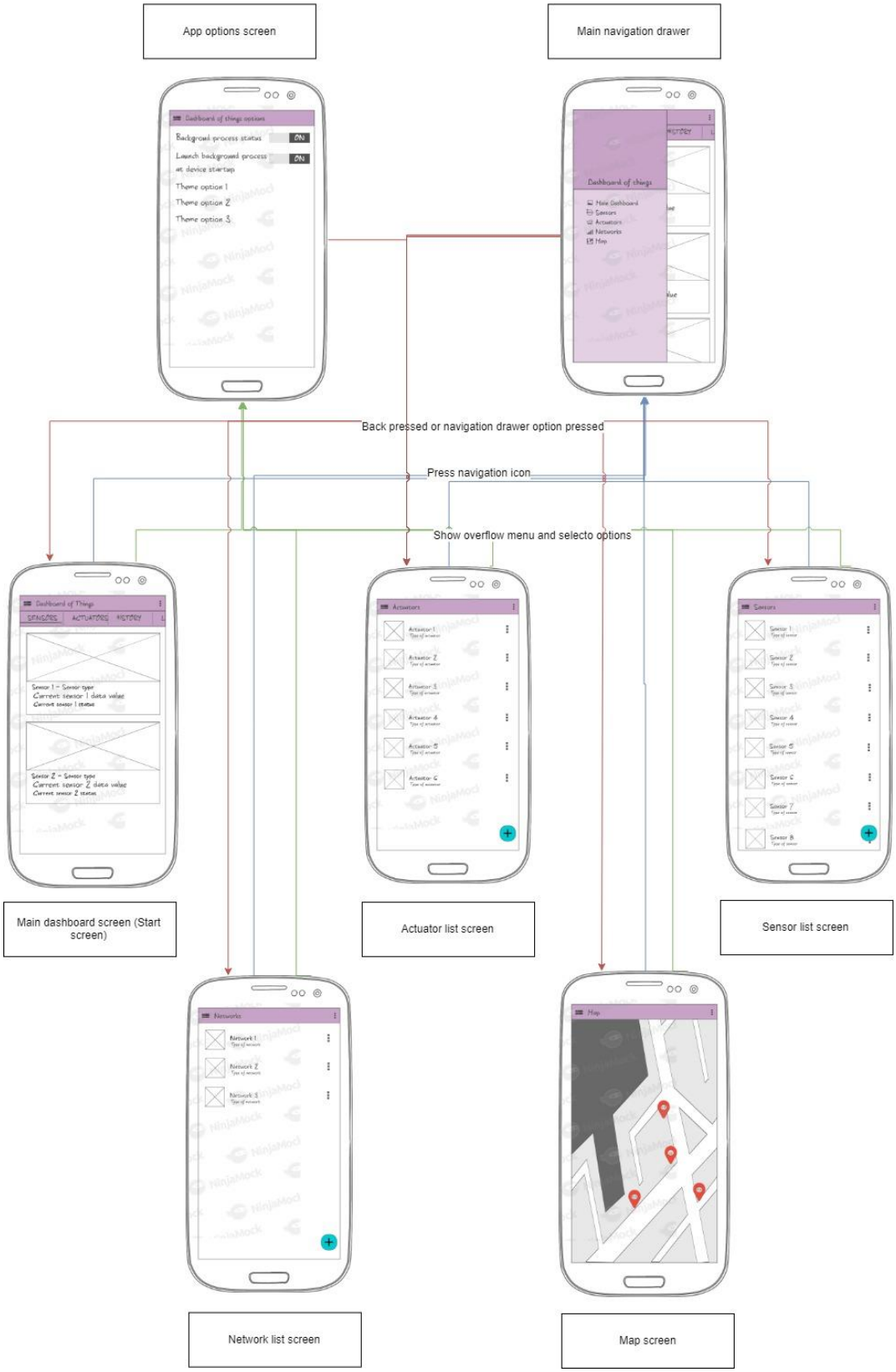
- Monitoring of IoT sensors, receiving and updating data collected in real time and showing historical data in different ways.

- Control of IoT actuators, being able to send commands to them in order to set their status.
- HTTP and MQTT protocols available for networking. IoT devices will use a configured network in order to get/set data. HTTP and MQTT protocols allows to use most of IoT cloud platforms.
- Support of XML, JSON and plain text for extracting data from messages received or building message for messages to be sent.
- Tracking of location of IoT devices through map. Location of an IoT device can be fixed or extracted from info received from backend.
- Simple and configurable user interface. Users will be able to configure the IoT devices showed at the main dashboard of the app, as such as general options about the theme of the app.
- A set of widgets for configuring your own dashboard in the launcher of your application. These widgets will include sensors, actuators and historical graphs.
- Alert system based on notifications to the user. The alerts will have warning and critical levels and will be configured through the use of thresholds.

User Interface Mocks

In this section we'll show the mockups thought for the application and the UI flow thought for the application.

Main navigation level



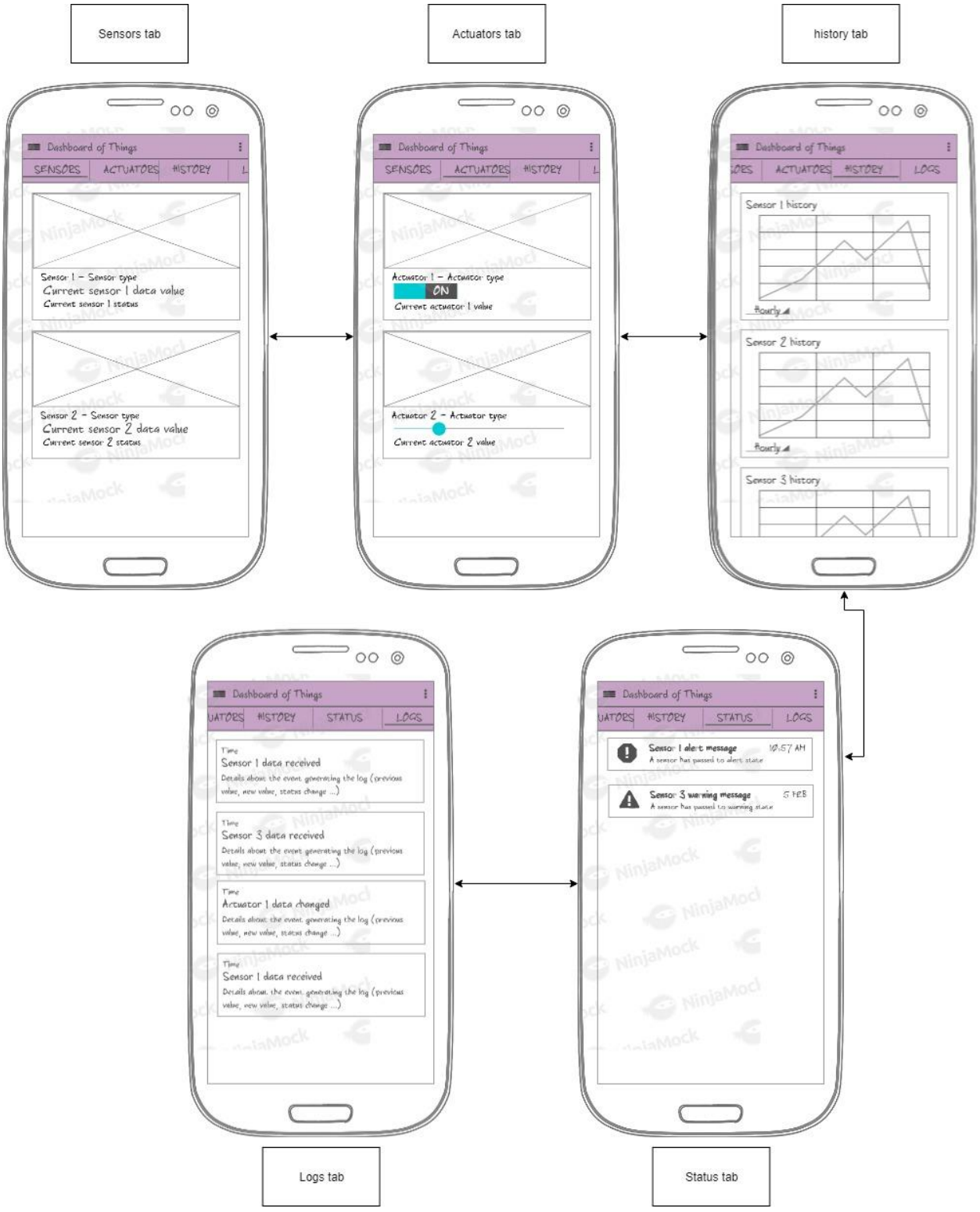
There will be 5 screens in the main level of navigation inside the application:

- Main dashboard: It’s the initial screen of the application. It shows a tabbed area where you can see both real time and historical data from the configured sensors, send commands to the actuators or see the active alerts or logs.
- Network list screen: It’s a screen for listing the networks (the backend servers to connect) configured in the application. From here you can add, edit and remove networks.
- Sensor list screen: It’s a screen for listing the IoT sensors configured in the application. From here you can add, edit and remove sensors.
- Actuators list screen: It’s a screen for listing the IoT actuators configured in the application. From here you can add, edit and remove actuators .
- Map screen: It’s a screen for showing the location of those sensors and actuators it’s location you have set. By selecting one you can go to the details of the sensor or the actuator.

For moving between these screens, a navigation drawer will be implemented, being shown by pressing the navigation icon at each of these screens.

Apart from the navigation icon, each screen has an overflow menu with an item for going to options screen, where you can configure the main options of the application.

Main dashboard screen

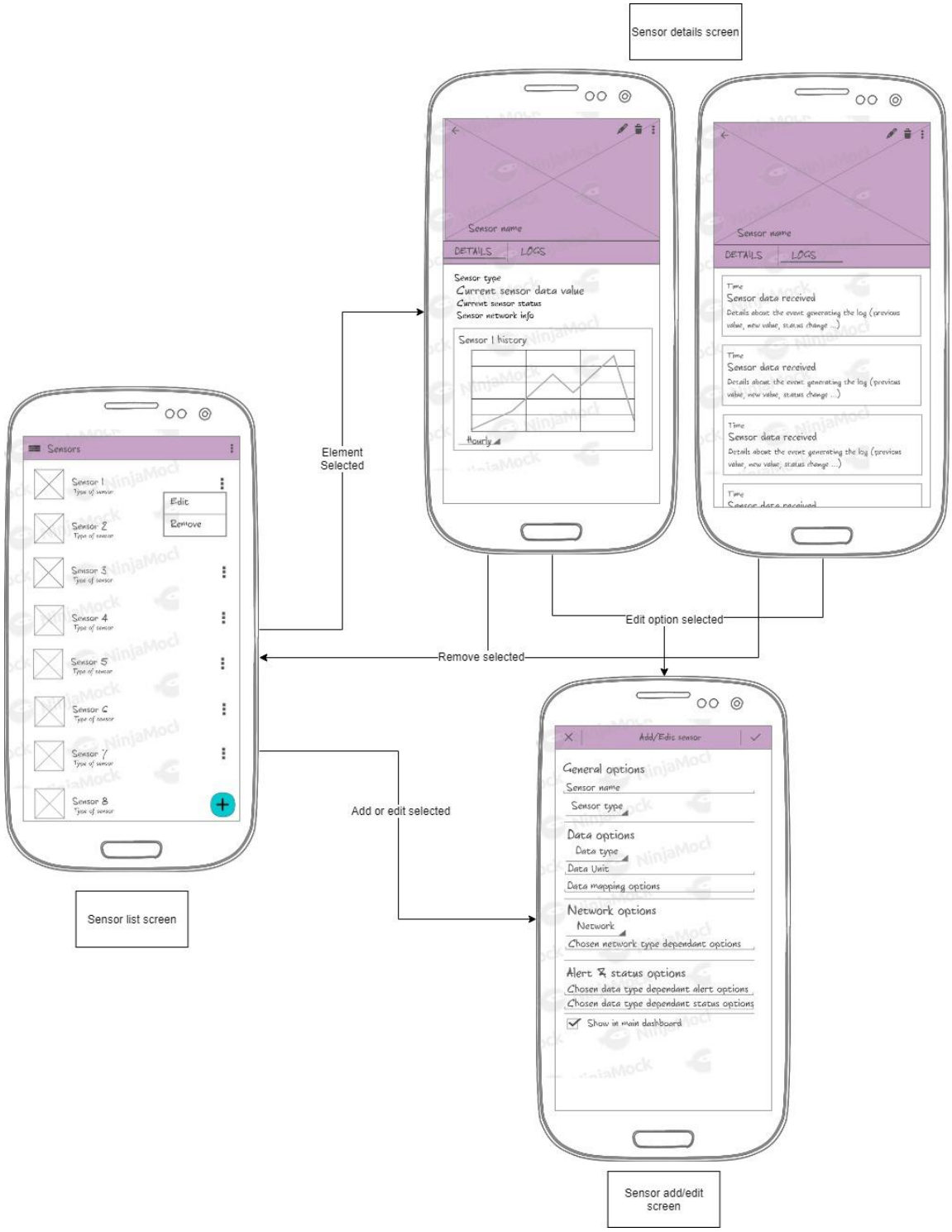


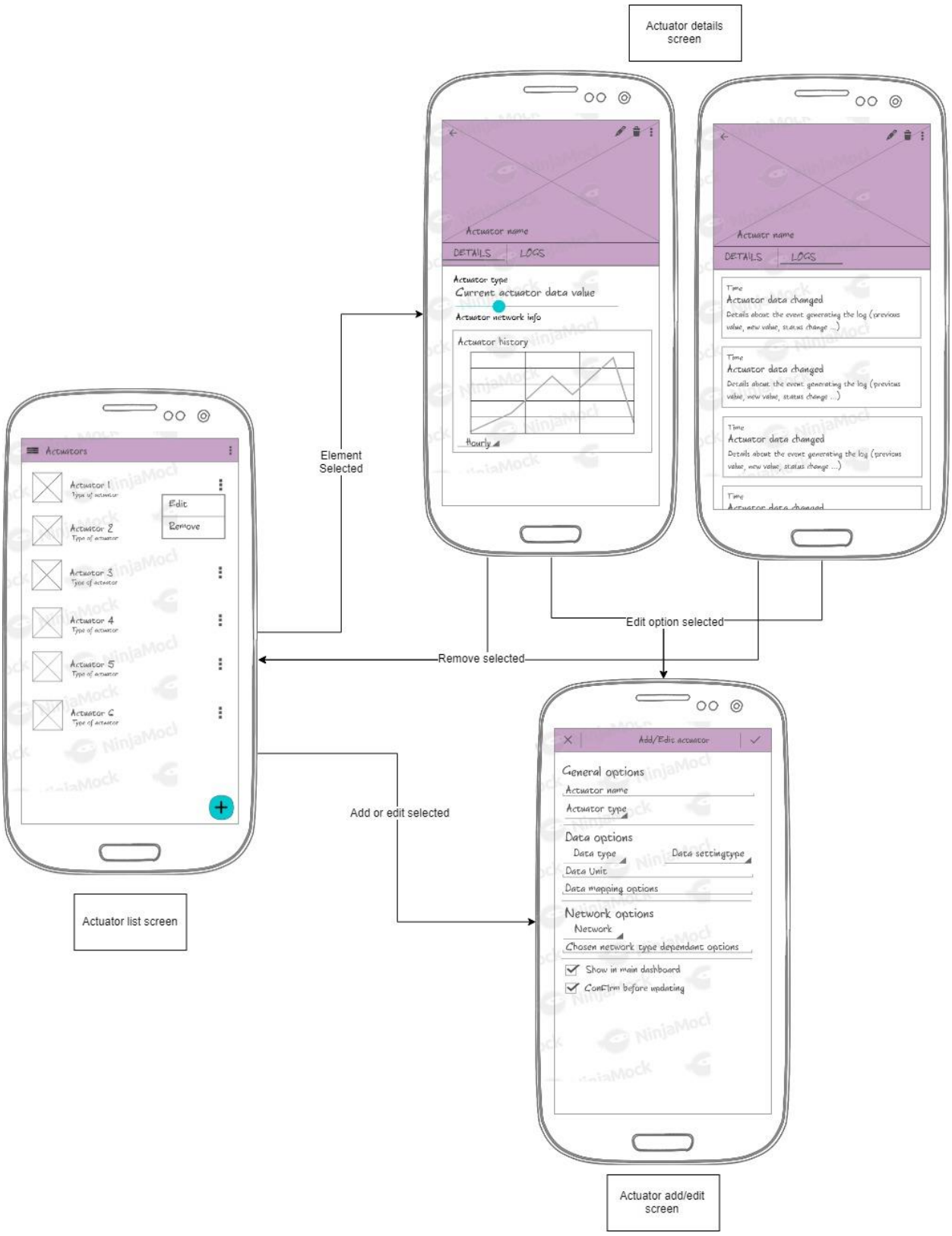
The main dashboard screen is the screen shown at first when app is launched. It contains a summary of the data and events related with the configured sensors and actuators. It’s composed of 5 pages:

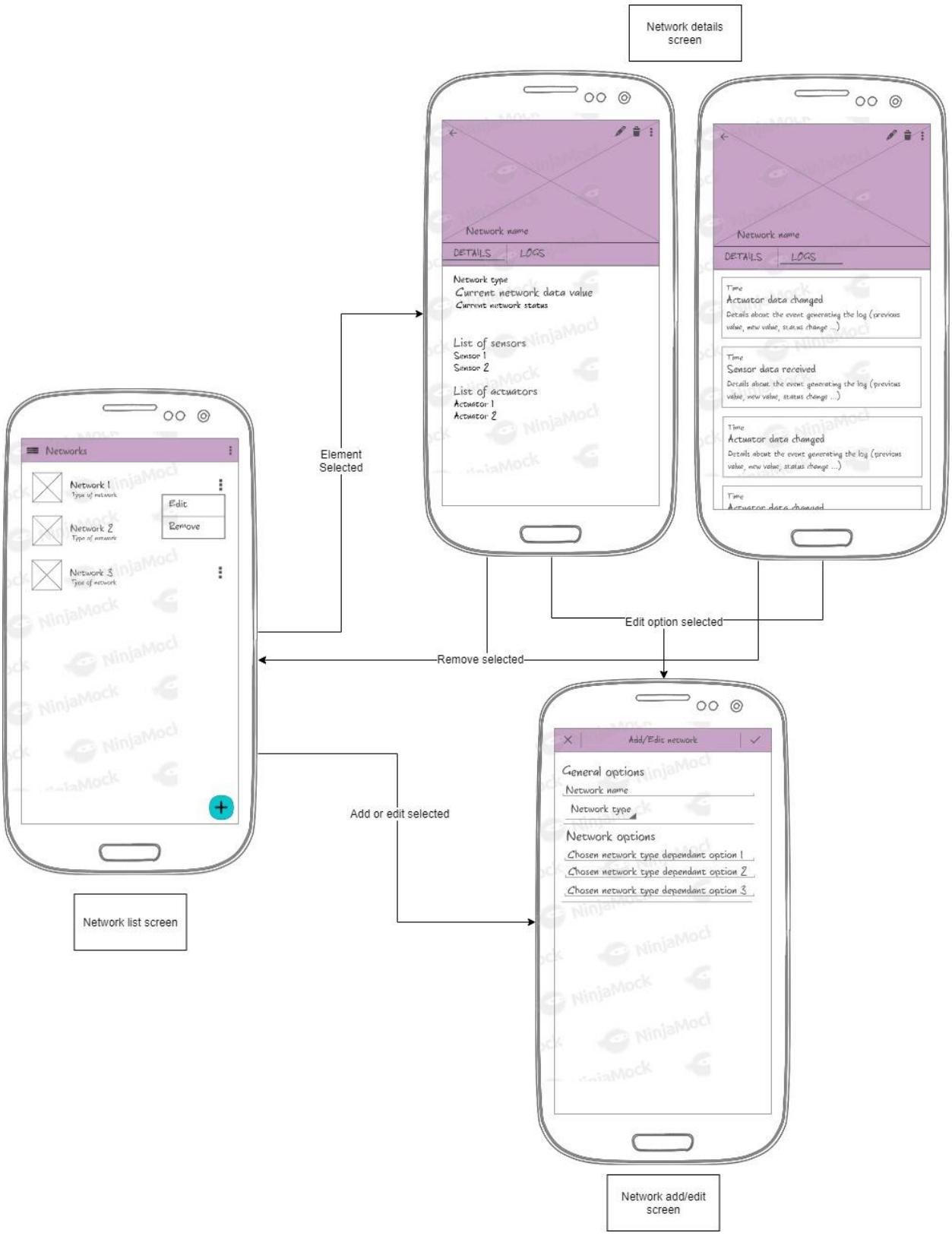
- Sensor page: with the current value and status of the configured sensors.
- Actuators page: allowing to send commands and set values of configured actuators.
- History page: with graphs containing info about historical data values of sensors.
- Status page: with a list of the alerts currently active.
- Logs page: with a log of the last events occurred in the application.

The navigation between these pages will be done through the use of tabs.

Networks/sensors/actuators management screens




Actuator details screen
Actuator name
DETAILS
LOGS
Actuator type
Current actuator data value
Actuator network info
Actuator history
Hourly
Time
Actuator data changed
Details about the event generating the log (previous value, new value, status change ...)
Element Selected
Edit option selected
Remove selected
Actuators
Actuator 1
Type of actuator
Edit
Remove
Actuator 2
Actuator 3
Actuator 4
Actuator 5
Actuator 6
Add or edit selected
Add/Edit actuator
General options
Actuator name
Actuator type
Data options
Data type
Data settingtype
Data Unit
Data mapping options
Network options
Network
Chosen network type dependant options
Show in main dashboard
Confirm before updating
Actuator list screen
Actuator add/edit screen


Network details screen
Network name
DETAILS
LOGS
Network type
Current network data value
Current network status
List of sensors
Sensor 1
Sensor 2
List of actuators
Actuator 1
Actuator 2
Time
Actuator data changed
Details about the event generating the log (previous value, new value, status change ...)
Time
Sensor data received
Details about the event generating the log (previous value, new value, status change ...)
Time
Actuator data changed
Details about the event generating the log (previous value, new value, status change ...)
Element Selected
Networks
Network 1
Type of network
Network 2
Type of network
Network 3
Type of network
Edit
Remove
Edit option selected
Remove selected
Add or edit selected
Add/Edit network
General options
Network name
Network type
Network options
Chosen network type dependant option 1
Chosen network type dependant option 2
Chosen network type dependant option 3
Network list screen
Network add/edit screen

There are three types of elements to be managed on the application:

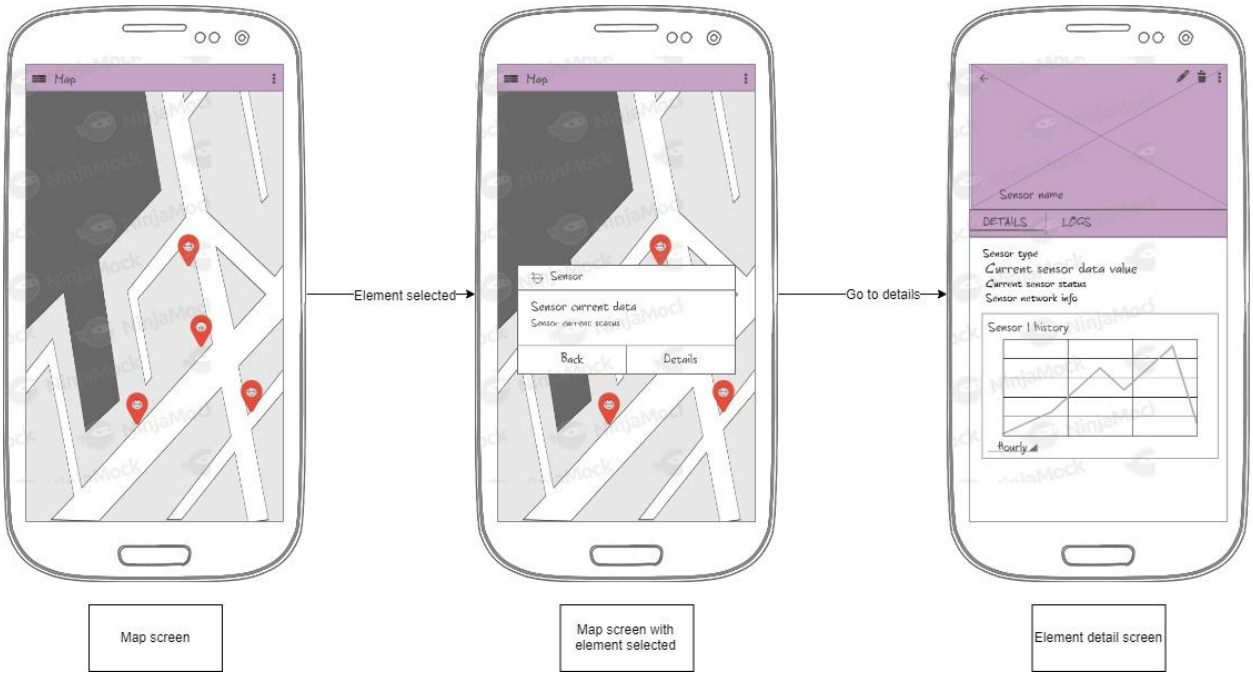
- Networks: connections to backend servers.
- Sensors: IoT devices providing data through a network.
- Actuators: IoT devices you can set data through a network.

Here is the UI flows of the three element management screens. They are pretty similar.

They are based on a master/detail flow pattern plus a third level which will be de edition screen:

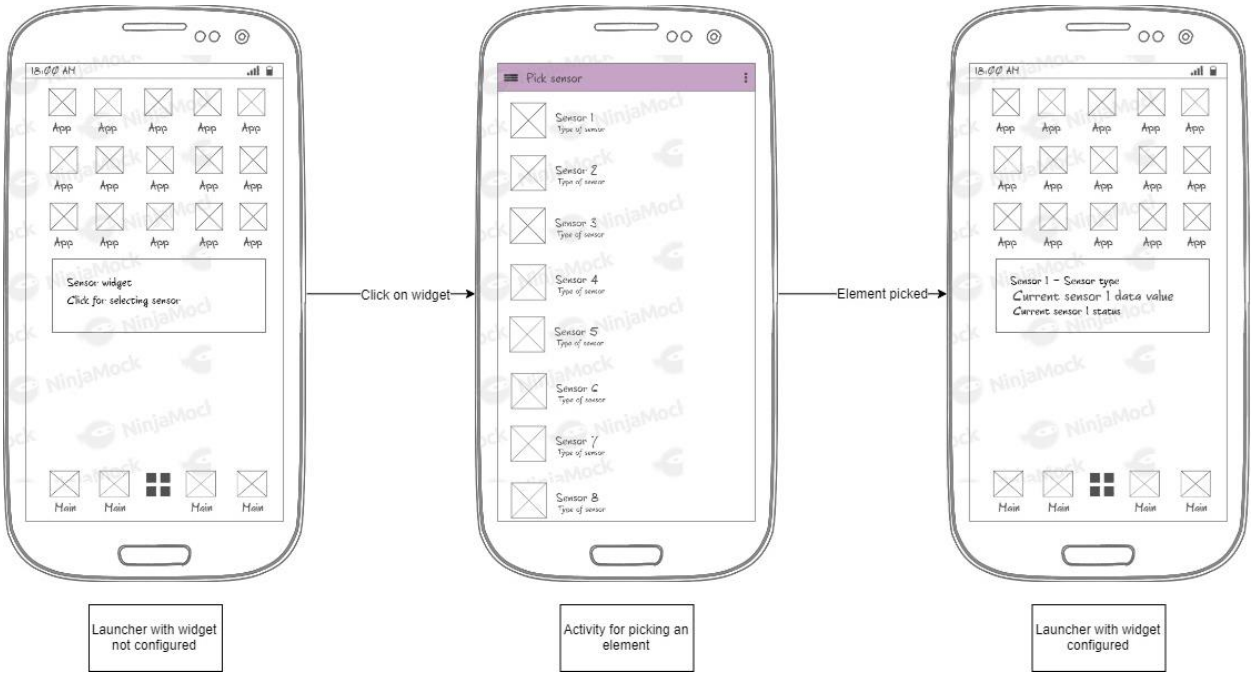
- In the list screen, you can see listed all elements configured. From here, you can add a new element (jumping to the edition screen in create mode), you can select a listed element (jumping to the details screen), you can select to edit an element (jumping to the edition screen in edit mode) or you can remove an element (removing it from DB after confirming it by a dialog, staying in the list screen).
- In the details screen, you can see the details about configuration and current status of the element. Here you have two tabs: the details tab is for showing the configuration and current status of the element; the logs tab is for showing the last log events related to the element. From here, you can edit the element (jumping to the edition screen in edit mode) or you can remove the element (removing it from DB after confirming it by a dialog and returning to list screen). Pressing back button will make you go back to list screen.
- In the edition screen, you can create or edit the configuration paramaters of the element. There is two modes of opening this screen: create mode, for creating a new element, or edit mode for editing an existing element. Once changes done, you can press the confirm button (for saving configured data to DB and return to previous screen) or you can press back button or cancel button (returning to previous screen without saving changes).

Map screen



The map screen will show all sensors and actuators with location set deployed on a map. You will be able to navigate through the map and look for them. If you click over an element deployed, it will show you the element identity, allowing you to navigate to the details page of the element.

Widget

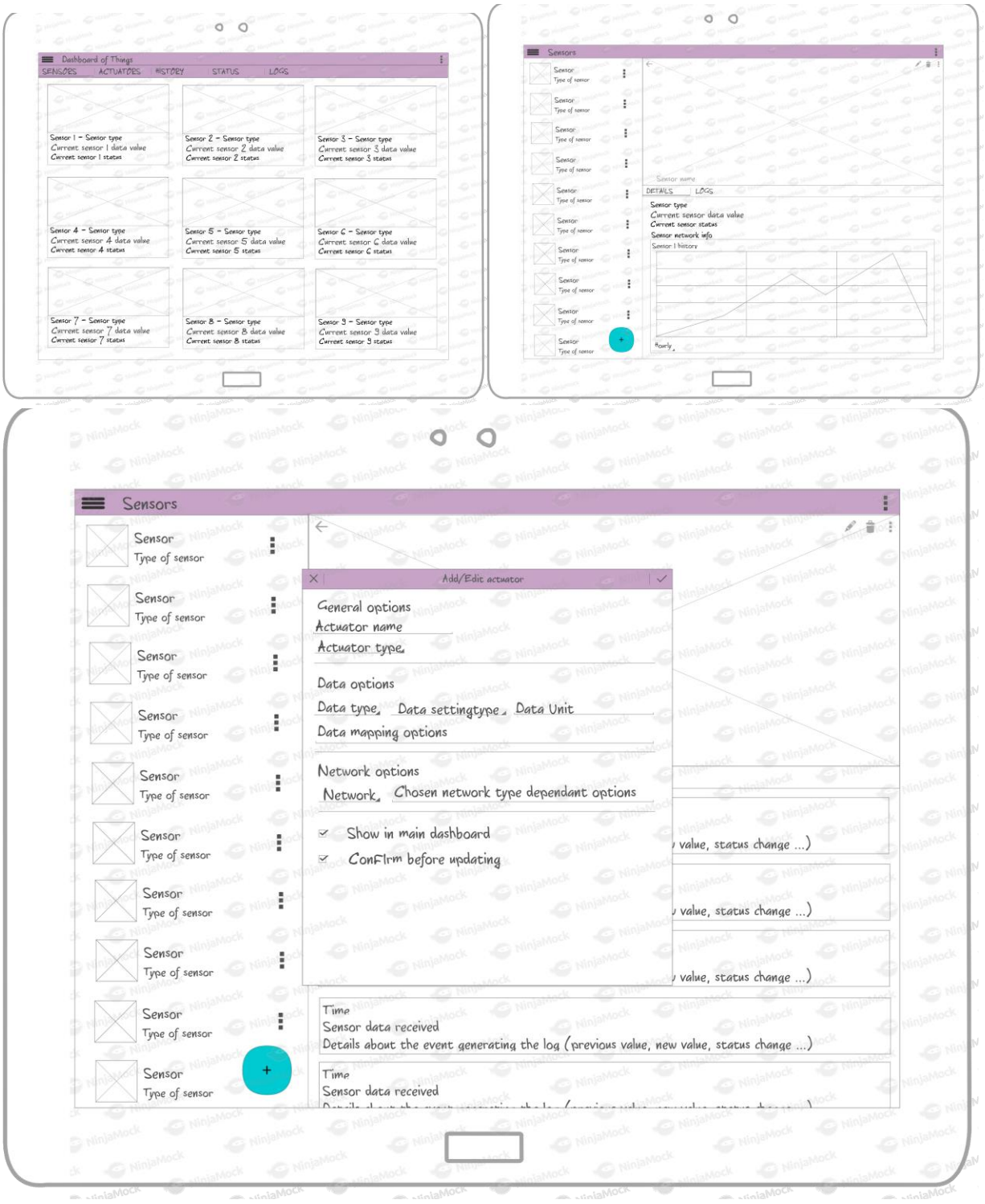


The application will provide a widget for showing real time info from a sensor in real time. If possible, widgets for managing actuators and showing history graphs about sensors will also be created.

When the widget is deployed, it’s not configured. You have to select a sensor to be attached to the widget.

For doing so, you have to click over the unconfigured widget. When clicking on it, an activity for picking the sensor to attach will be launched. Once selected a sensor, the widget will be configured and will start to show the last data received from the sensor.

UI in tablet



The UI flow in tablet devices will be the same as in phone devices, but with some differences. First, in the dashboard screen pages we'll use the more space available in screen for showing the listed info in the form of a grid with several columns, depending on screen width.

Second, we'll take advantage of the master/detail flow pattern for showing both the element list and the element details screen in the same joint screen, using the left side for the list and the rest for the details.

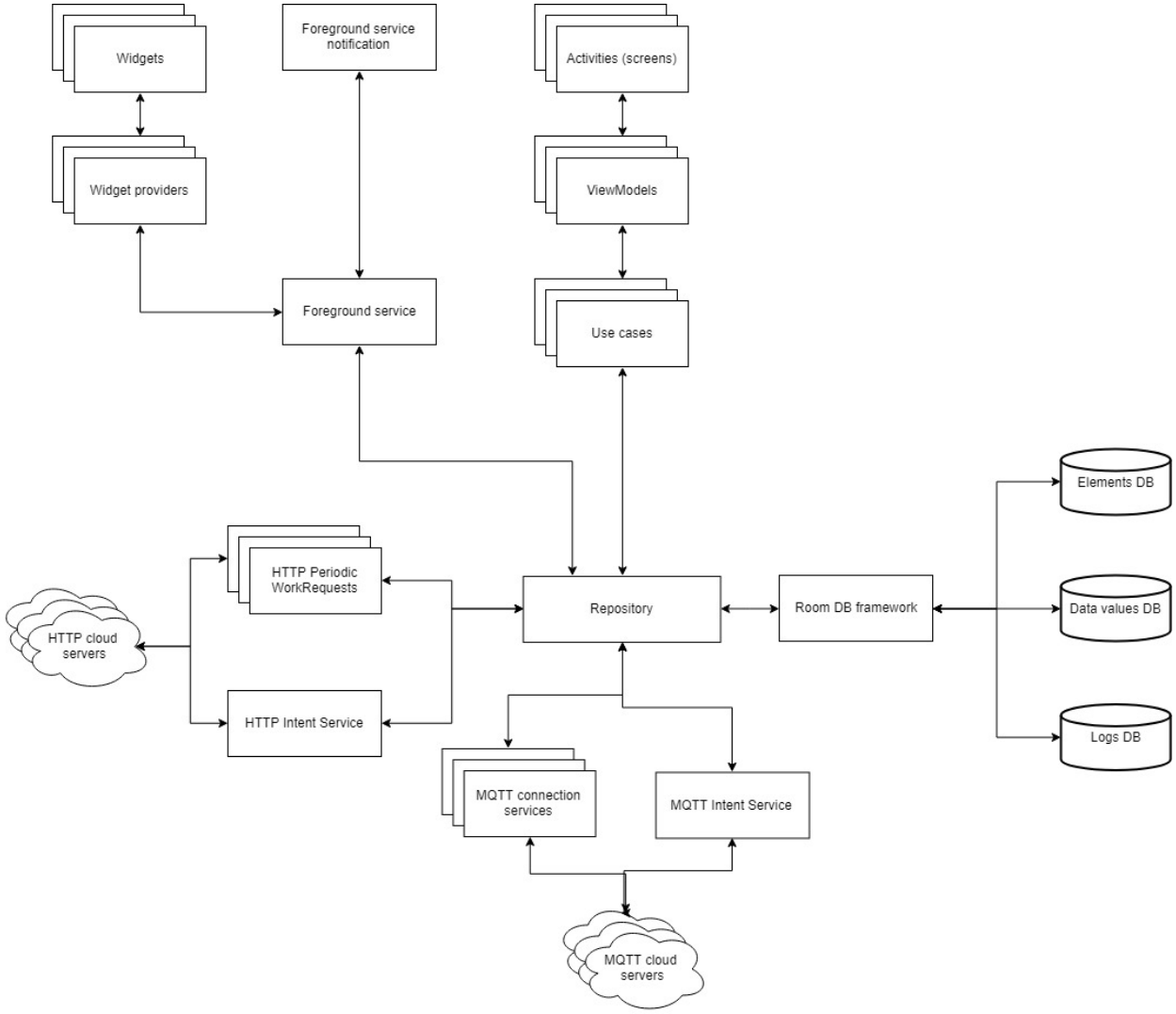
Third, the add/edit element screen will be launched as a dialog.

Key Considerations

How will your app handle data persistence?

App architecture

Below is the architecture proposed for the application:



The application will have an architecture based on the recommended architecture for Android applications since the addition of Android Jetpack libraries and the addition of use cases objects for having the application architecture complies with the clean architecture pattern principles.

The activities will be in charge of dealing with UI, updating it or getting events from it. The ViewModels will be used as an interface between UI and application logic. The use cases will deal with the application logic in charge to decide which data to get or update. And finally the repository will provide an interface for accessing the database and the backend servers.

All data used by the app will be managed from a repository object, providing requested data by using LiveData objects without caring about the origin of the data. The repository will obtain the data from either a database of from cloud backend servers.

The repository will be created and managed by a foreground service which will be started or stopped from the app settings screen. In the same settings screen it will be also possible to configure the service to be started at Android device boot. The foreground service will be also in charge of dealing with transmitting and receiving info from the widgets.

Entities managed

Here is a list of the entities managed by the application:

- Network: A network is a connection to a cloud backend server. It can be HTTP (datagram oriented protocol standard in REST APIs) or MQTT (connection oriented protocol standard in IoT environments). For configuring a network, we need to manage the following parameters:
    - Standard parameters: Name, type, image from Android device to be used for representing it …
    - Network-specific parameters: type of network, base URL, port, authentication and security options …
- Sensor: A sensor is an IoT device providing you with info. Examples are a thermometer, a movement detector or a heart rate tracker. The data provided can be a boolean value (active or not), an integer value (limited), a decimal value (limited), a string or an element between a list of configured values. A sensor has to be related to an existing network. For configuring a sensor, we need to manage the following parameters:
    - Standard parameters: Name, type, image from Android device to be used for representing it …
    - Network parameters: network to use, and parameters associated with that network (relative URL where to request data and periodicity for requesting it in case of HTTP; topic filter to subscribe for requesting data in case of MQTT).
    - Message parameters: parameters for processing the message received in order to get info. Whether it's a plain text, XML or JSON message; in the case of plain

text limiting strings in order to find the data; in the case of XML and JSON specifying the node path to use for finding the data.
    - Data parameters: type of data (boolean, integer, decimal, string, list), configuration depending on the type of data, configuration for mapping data received for showing it (like unit to be used).
    - Alert parameters: configure above and below thresholds for launching warning and critical alerts.
    - Location parameters: configure a fixed location or how to get the location coordinates from message.
    - Whether to show data in main dashboard.
- Actuator: An actuator is an IoT device you can send commands. Examples are a thermostat, a light and a smart plug. The command will be a message with data attached, being able that data to be a boolean value (activate or not), an integer value, a decimal value, a string or an element between a list of configured values. An actuator has to be related to an existing network. For configuring an actuator, we need to manage the following parameters:
    - Standard parameters: Name, type, image from Android device to be used for representing it …
    - Network parameters: network to use, and parameters associated with that network (relative URL where to send request and request type in case of HTTP; topic to publish in case of MQTT).
    - Message parameters: parameters for creating the message which will be used as command. The message will be composed using a template to be defined by the user specifying where to put the data to send, between other parameters.
    - Data parameters: type of data (boolean, integer, decimal, string, list), configuration depending on the type of data, configuration for mapping data set to data to be sent.
    - Location parameters: configure a fixed location if wanted for the actuator.
    - Whether to show data in main dashboard.

Database data

For the management of general app configuration options Shared Preferences will be used.

For the rest of data stored by the application Room framework will be used, storing data used by the application in databases managed by this library.

This is the list of databases used in the app:

- Elements database: database with tables containing the configuration info used for setting up networks, sensors and actuators.
- Data values database: database where to save the values received from IoT sensors and the time the info was received.

- Logs database: database where to store logs about every event that happened in the system.

Network data

The different methods for getting data from cloud backend servers will differ depending on the type of network.

- In the case of HTTP networks, data is got by sending GET requests to the server and extracting the data from the response message. The application will allow to configure periodical requests or will allow to request a update of the data at any time.
    - Periodical requests will be managed by using PeriodicWorkRequests and the Work Manager. For each IoT sensor using HTTP network and with periodical update activated a PeriodicWorkRequest will be scheduled. These WorkRequests will send a GET request to correspondent backend server and will extract the outcome. The outcome of the request will be transmitted to the repository by using RxJava library.
    - Non-periodical request will be made by using an IntentService receiving orders for making GET requests and extract the outcome. The outcome of the request will be transmitted to the repository by using RxJava library.
- In the case of MQTT networks, the backend server is a MQTT broker we have to be connected in order to get messages from topics we're subscribed. These topics will represent the IoT sensors configured to use this network.
    - In these cases a service will be launched for handling the communication with the MQTT broker. This service will be in charge of handling the reception of messages in a background thread and notify it through a callback. In the callback the output will be extracted and transmitted to the repository by using RxJava library.

The different methods for setting data from cloud backend servers will differ depending on the type of network.

- In the case of HTTP networks, a message and a URL will be composed from the data to be set, the network configuration and another elements like current timestamp. This data will be sent using PUT or POST requests to the server. The sending of the request will be done by putting a send order to the same IntentService used for the sending of not-periodical GET requests. The outcome of the request will be transmitted to the repository by using RxJava library, for updating actuators historical data.
- In the case of MQTT networks, a message and a topic will be composed from the data to be set, the network configuration and another elements like timestamp. The service lifted for managing the connection to the MQTT broker can send the defined message to the broker using the defined topic. This will be done by putting a send order to a MQTT IntentService dedicated to MQTT sendings. The outcome of the

sensing will be transmitted to the repository by using RxJava library, for updating actuators historical data.

Describe any edge or corner cases in the UX.

Here is a list of possible edge cases currently identified in the UX flow and how to deal with them:

- Misconfiguration of parameters when creating or editing networks: apart from limiting modification of these parameters in order to avoid mistakes in configuration, any problem appeared during connection will be logged and reported to the user through notifications and in the log screens.
- Misconfiguration of parameters when creating or editing sensors and actuators: the same with sensors applied to sensors and actuators, any problem creating, parsing or sending messages to backend servers will be reported to the user through log screens and notifications.
- Remove a network with linked sensors and actuators. The removal of a network will suppose the removal of all sensors and actuators linked, so as network logs
- Remove a sensor or actuator with historical data linked: this will suppose the removal of the historical data and the logs related to it.
- The user wants to configure too many entities, consuming a lot of storage: the number of networks will be limited to 20, while the total number of sensors and actuators will be limited to 100.
- Too many time storing data values of a sensor can consume a lot of storage: the number of data values will be limited to 10000 per sensor.
- Too many logs can consume a lot of storage: the number of logs will be limited to 2000. With the previous three elements, we limit the storage consumed by the application to approximately 600 MB.
- A sensor or an actuator with a widget deployed is removed. In this case the widget will be resetted to initial state, requiring the selection of a sensor or an actuator.

Describe any libraries you'll be using and share your reasoning for including them.

Here is a list of the external libraries to be used on the app:

- Dagger2: Injection dependency library. It will be used for the injection of objects along the application in order to get a more modularized app architecture.
- RxJava2: Reactive programming library. It will be used for notifying to the repository the result of background tasks done for getting data from cloud backend servers.
- Androidx Room: Persistence library. It will be used for the creation and management of the database containing the elements configured and the historical data.

- Androidx Lifecycle: Library with utilities for managing Android elements lifecycle and based on reactive programming. It will be used for getting in the UI change notifications done by the repository about data being shown.
- Androidx Data binding: UI data binding library. It will be used for managing the updating of the UI with the data to be shown.
- Androidx Work Manager: background job scheduling library. It will be used for scheduling periodical update of IoT sensors using HTTP networks.
- Volley: HTTP network library. It will be used for the management of HTTP requests and responses when using a HTTP network.
- Eclipse PAHO Android service: MQTT client library. It creates a service in charge of publishing messages to a MQTT broker or receiving messages from a MQTT broker subscribed topic, all this in a background thread. It will be used for the management of MQTT messages when using a MQTT network.
- Glide: Image loading management library. It will be used for the loading of images linked to sensors, actuators and networks, shown in the list and in the details of these elements.
- MPAndroidChart: Chart library. It will be used for the creation of line charts for showing historical values of a IoT sensor data.
- Google Play Services Maps: Google maps management library. It will be used for the creation of a screen showing the location of all devices configured.
- Google Play Services Admob: Google Admob management library. It will be used for showing in several points of the application ads in order to monetize the application.

Describe how you will implement Google Play Services or other external services.

The app will use two Google Play Services: Google Maps and Google AdMob.

- Google Maps will be used for showing the location of the IoT devices around the world. This location will be configured as fixed or will be extracted from a network message.
- Google AdMob will be used for showing ads during the execution of the application. There will be shown full-screen interstitial ads each time the app is open and every 4 times a main screen is open from navigation drawer.

Next Steps: Required Tasks

This is the section where you can take the main features of your app (declared above) and break them down into tangible technical tasks that you can complete one at a time until you have a finished app.

Task 1: Project Setup

First of all, we have to create and setup the project to start developing. This task can be structured in the following subtasks:

- Create project.
- Configure project libraries.
- Configure skeleton of first activity.
- Configure permissions.
- Configure certificate for signing the APK.
- Configure gradle tasks for automatically generate a signed APK.
- Configure Dagger classes for injecting dependencies in future classes.

Task 2: Implement entities classes

We have to create the classes for managing the different entities used by the application.

- Implement network configuration class.
- Implement sensors configuration class.
- Implement actuators configuration class.
- Implement historical data values class.
- Implement log class.

Task 3: Implement database framework

We have to create a framework for managing the databases created for storing user data.

- Design and create an elements DB, with tables for networks configuration, sensors configuration and actuators configuration.
- Design and create data values DB, with tables for historical data from sensors and historical data for actuators.
- Design and create logs DB, with a table for storing log events.
- Design the interface for doing CRUD operations over the database tables using the entity classes created before.

Task 4: Implement HTTP and MQTT management classes

We have to create the described classes for managing the connections to the backend servers.

- Implement HTTP IntentService.
- Implement MQTT IntentService.

- Implement wrapper for deploying MQTT services offered by eclipse PAHO library.
- Implement PeriodicWorkRequests for HTTP and a wrapper for scheduling them.

Task 5: Implement data repository

We have to create a repository for managing the now created database and network databases.

- Create repository class.
- Populate it with methods for managing database tables (creation, updating, getting and removal of networks, sensors and actuators configuration, so as historical data values and logs). Create compound entity classes for that if necessary. These functions will use LiveData objects.
- Populate it with methods for starting and stopping cloud HTTP and MQTT backend connections, and configuring HTTP periodical updates.
- Create a reactive system with the use of RxJava for getting and processing backend messages received and store it in database.

Task 6: Implement alert system

We have to create a system for sending alerts based on notifications.

- Create a builder class for deploying notifications representing the alerts.

Task 7: Implement use cases

We have to create a set of use case classes for managing the actions to be done between the UI and the logic of the application.

- Create use case classes for elements configuration management (CRUD operations).
- Create use case classes for managing the obtention of collections of lists of historical data values.
- Create use case classes for managing the obtention of status of elements.
- Create use case classes for getting current alerts and the logs.
- Create use case classes for checking if a new data received has to create new alerts, logs or has to be notified anywhere else in the app.

Task 8: Implement foreground service

We have to create a foreground service for managing the repository, starting it (and HTTP and MQTT connections with it) when required. It will be also used for managing the created widgets.

- Create the service class.
- Implement the foreground notification.
- Implement the init of the repository.
- Implement a system for starting or stopping this service from a use case.

Task 9: Implement main dashboard screen

We have to create the main dashboard screen.

- Implement the ViewModel for the Activity.
- Implement the XML layouts for the activity.
- Implement the Activity and register it in the manifest.
- Implement the XML layout for the dashboard pages.
- Implement the Fragments for the dashboard pages.
- Implement the navigation drawer.

Task 10: Implement element management screens

We have to create the element management screens.

- Implement the ViewModels for the network list activity, the network details activity and the network edition activity.
- Implement the XML layouts for these activities.
- Implement these activities and register them in the manifest.
- Implement the XML layouts and Fragments for the details and logs pages in the network details activity.
- Configure the network list activity to be launched from the navigation drawer.
- Repeat these steps for both sensors and actuators.

Task 11: Implement map screen

We have to create the map screen.

- Implement the ViewModel for the map Activity.
- Implement the XML layouts for the map activity.
- Implement the map Activity and register it in the manifest.
- Implement the deployment of markers on the map.
- Implement the notifications for the details of the elements related to the markers to be launched when clicked. Implement the launching of element details when required.

Task 12: Implement sensor widget

We have to create the sensor widget controlled by the app.

- Implement the widget provider.
- Implement the XML layout of the widget.
- Implement the management of widgets from the service.
- Implement the activity to configure the widget.

Task 13: Implement options screen

We have to create the app configuration screen.

- Implement the ViewModel for the Activity.
- Implement the XML layouts for the activity.
- Implement the Activity and register it in the manifest.
- Implement the main navigation level activities for launching the screen.