Architecture Document

The smart home project

Cronus

Version 7.0

Client: M.F. Lungu, A. Karountzos

C. Ausema,
S. Evanghelides,
L. Holdijk,
M. Helmus,
S. Maquelin,
F. te Nijenhuis,
F. Ritsema,
S. Zijerveld

Introduction

Nowadays, if you want to turn on a light in your house, you have to look for the light switch in the dark. If you leave work early, you will come home to a cold house. Everyday processes like unlocking the door, and controlling the lights will be automated by Hestia with just the simple touch of a button. The project goal is to develop an easily extendable, secure application that makes the life for residents easier.

In accordance with object oriented design principles, we decided to split the program into three separate parts, namely a client, a server and the peripherals. We structure our program in such a way that peripherals can be added and removed from the server by pushing a single button.

This document describes in detail how the system works and the reasons why certain design decisions were made. For the convenience of the reader, a glossary was added in the appendix, as this document contains some naming conventions specific to this project.

General overview of entire system

The Hestia platform/system contains three main components;

- A client, which for now will be an android application
- The peripherals, which can perform home automation tasks. An example of this would be a Philips Hue light bulb.
- A server, which provides a general interface for communication between the client and the peripherals.

In figure 1, a schematic overview can be found.

Client

The app is a so called thin-client. This means that most of the computation is performed on the server, and the client is simply a wrapper around the data returned by the server. The client interacts with the server by sending HTTP requests over a HTTPS connection according to the REST principles. The REST API provided by the server is used for this. The payloads of these messages are JSON objects. Using these messages the client can retrieve a list of peripherals and the actions these peripherals can perform.

For the MVP, the login screen is a mockup. To login, the user enters “admin” as the username and “password” for the password by default. The remember me function works, so the credentials have to be entered only once. This is in accordance with our guidelines of making the software as accessible as possible. It is also possible to change the credentials.
After logging in, the app moves from the LoginActivity to the HomeActivity. An Activity is a term from the Android framework used to describe a window in which the User Interface (UI) can be placed in the app, on which the current list of devices will be shown.
From this point, the user can add a new device, remove a device, change the state of the current devices or the name of a device. The above operations were achieved with the help of the REST API, by performing HTTP requests. (more information can be found on page 5).

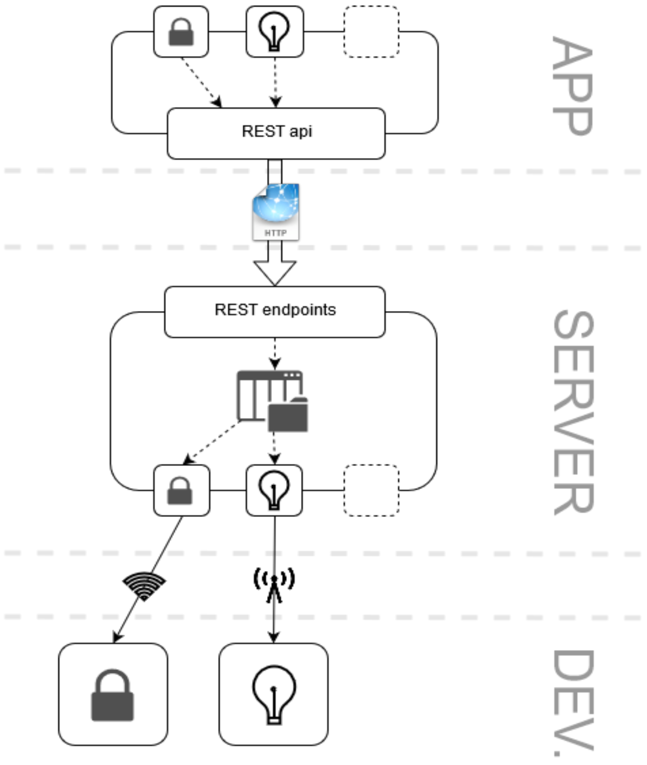


Figure 1: schematic overview of entire system

Server

The server operates as a REST API[1]. Meaning that it can be interacted with based on the HTTPS methods GET, PUT, POST and DELETE.

The server will keep track of the different peripherals that the client can communicate with. When the client requests a peripheral to perform an action the server needs to relay this message to the corresponding peripheral. To offer the possibility of extending the home automation system further the server uses a basic plugin system. The plugins in this system define how communication with a peripheral should be handled by the server.

Devices

The Hestia platform will mostly use third-party devices. To connect a third-party device to the Hestia platform a plugin can be created. This procedure is explained in more detail below.

Besides the third-party devices, the customer is still debating about developing a remote control motor in house. For the time being, the development of the motor has however been placed on hold.

[1] More on REST API: https://en.wikipedia.org/wiki/Representational_state_transfer

Client

Architectural overview

The Hestia app consists of a backend (package `backend`) and a frontend (package `UI`). The backend is concerned with the serialization of data into JSON objects, and the deserialization of responses from the server. The frontend is concerned with the presentation of this data to the user. The backend and the frontend communicate with each other through Android AsyncTasks (ATs). The frontend creates different ATs to perform various tasks, and each AT uses code from the backend to interact with the server. The frontend consists of multiple Android Activity classes, together with their subcomponents. The backend contains code for interacting with the server, storing the IP address, port number and the list of devices, along with their activators.

Backend

Regarding the backend part, the backend package contains the `exceptions` and the `models` packages, as well as the NetworkHandler and the ServerCollectionsInteractor classes. The NetworkHandler contains methods for executing the four HTTPS requests: GET, POST, PUT and DELETE. The GET request is used to retrieve information to the server, such as the list of devices or the plugins available in the Server. The POST request is used to send information to the server for adding a device or changing the current state of a device to a new one. The PUT request is used to change the name of the Device, while the DELETE request is used for removing a device from the server.

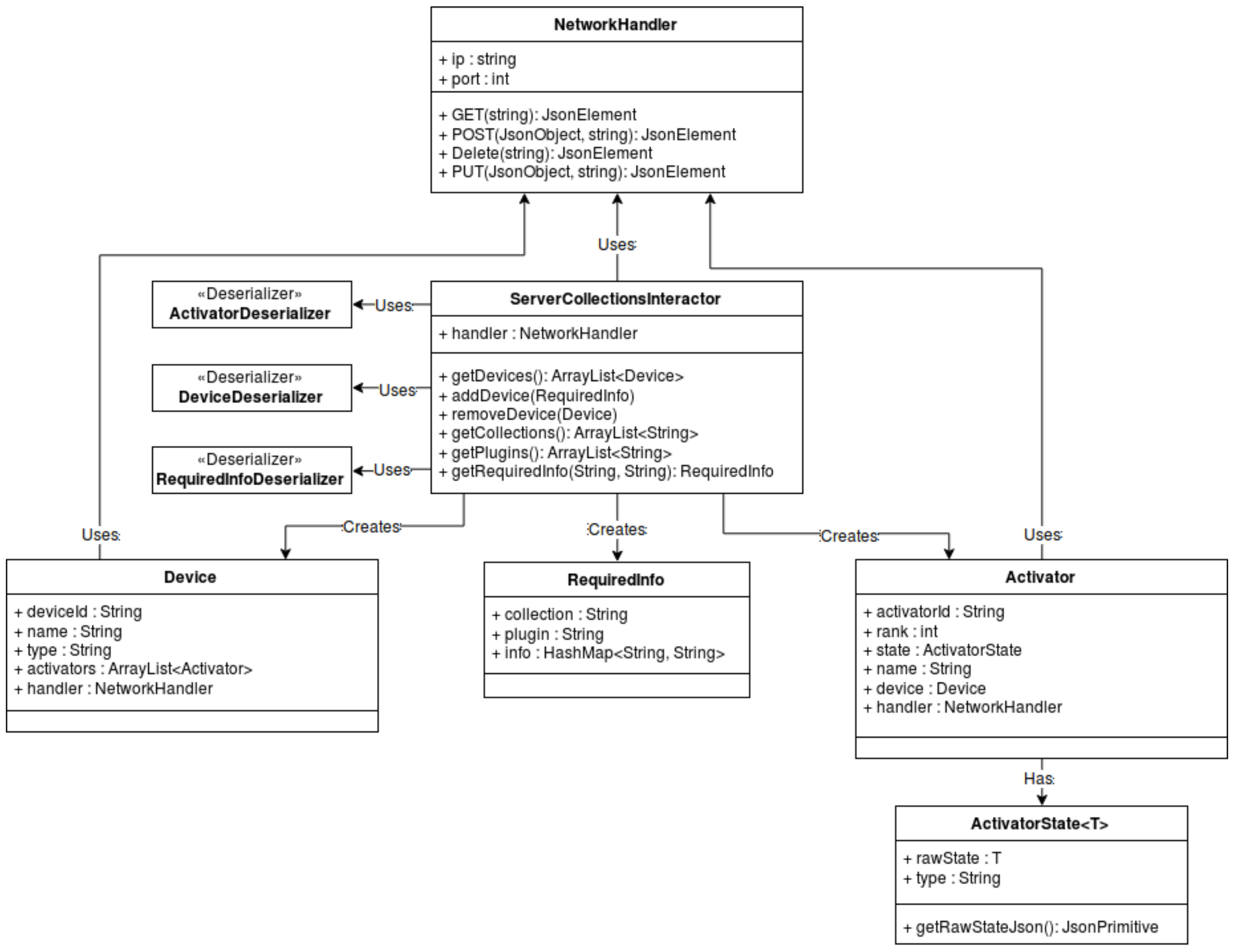
The NetworkHandler also contains methods for connecting to the server and getting and parsing the payload received from the server. The ServerCollectionsInteractor acts as a façade to the ATs performing different actions such as adding or removing a new device, changing the name of a device, getting the list of devices, collections or plugins. At the same time, exceptions are thrown if the server responds with an error to the requests from the Client.

The information sent or received by the Client concerns the classes inside of the `models` package, namely RequiredInfo, Devices and their Activators. This information is retrieved from the server as a JSON object. Google's GSON library is used for deserializing the JSON object into the aforementioned objects. Thus, inside of `models` package, there is the `deserializers` package which contains RequiredInfoDeserializer, DeviceDeserializer and ActivatorDeserializer, each used to parse the objects with the same name. They are needed because these are custom, rather complex objects, which cannot be deserialized using GSON's standard deserializers.

The RequiredInfo contains the chosen collection and plugin name, as well as the additional information, specific to each device (for instance, the bridge IP and port number of a Philips Hue

Light). The Device class contains the id, name, type, the list of activators and an instance of the current NetworkHandler. The Activator class contains the id, rank, name, state and the instance of NetworkHandler used in the Device class. The NetworkHandler will be used to directly execute a POST request when the name of a Device is changed. Same holds when the state of an Activator is changed. The state is represented in the ActivatorState class, which contains the type of the state (‘bool’ for toggle switches or ‘float’ for sliders) and the actual value (more information about Devices and Activators can be found in the Appendices).

The diagram below show how the ServerCollectionsInteractor, Device and Activator classes all use the NetworkHandler to notify the server of changes made in them by the front end.


NetworkHandler
+ ip : string
+ port : int
+ GET(string): JsonElement
+ POST(JsonObject, string): JsonElement
+ Delete(string): JsonElement
+ PUT(JsonObject, string): JsonElement
Uses
«Deserializer»
ActivatorDeserializer
Uses
ServerCollectionsInteractor
+ handler : NetworkHandler
«Deserializer»
DeviceDeserializer
Uses
+ getDevices(): ArrayList<Device>
+ addDevice(RequiredInfo)
+ removeDevice(Device)
+ getCollections(): ArrayList<String>
+ getPlugins(): ArrayList<String>
+ getRequiredInfo(String, String): RequiredInfo
«Deserializer»
RequiredInfoDeserializer
Uses
Uses
Creates
Creates
Creates
Uses
Device
+ deviceId : String
+ name : String
+ type : String
+ activators : ArrayList<Activator>
+ handler : NetworkHandler
RequiredInfo
+ collection : String
+ plugin : String
+ info : HashMap<String, String>
Activator
+ activatorId : String
+ rank : int
+ state : ActivatorState
+ name : String
+ device : Device
+ handler : NetworkHandler
Has
ActivatorState<T>
+ rawState : T
+ type : String
+ getRawStateJson(): JsonPrimitive

Frontend

While the frontend of the client application mainly consists of basic java functionalities, relying heavily on the pre-programmed Android classes and structure. Due to the clear distinction and the low coupling between the front and the backend classes it should be possible to change the frontend easily without changing the backend. This should make the transition to other platforms, such as IOS, relatively painless.

Currently the Android application has two different activities, one login activity and one home activity.

LoginActivity

The login activity is used as a basic layer of protection. It is not meant to provide full security to our system and we are aware of the limitations of the current implementation. It is only meant as a way of protection when the Android device is lent out by the owner to a trusted friend or family member.

When the user opens the application, he or she is prompted with a login screen. On the login screen the user is required to enter a username and a password. When the user does not want to be asked to enter this every time the application is started, he or she can decide to let the application remember that the login was successful. Before entering the credentials, the user has to enter the Server’s IP address. Pressing the button “Server” will display the DiscoverServerDialog which will trigger the automatic server discovery process (more details about this process can be found in the next sections).

HomeActivity

Within the HomeActivity the user can interact with the devices installed on their server. The HomeActivity class has a DeviceListFragment to hold these devices. Besides this, the home activity also features a toolbar that can be used to change the ip, change the username and password, or simply logout.

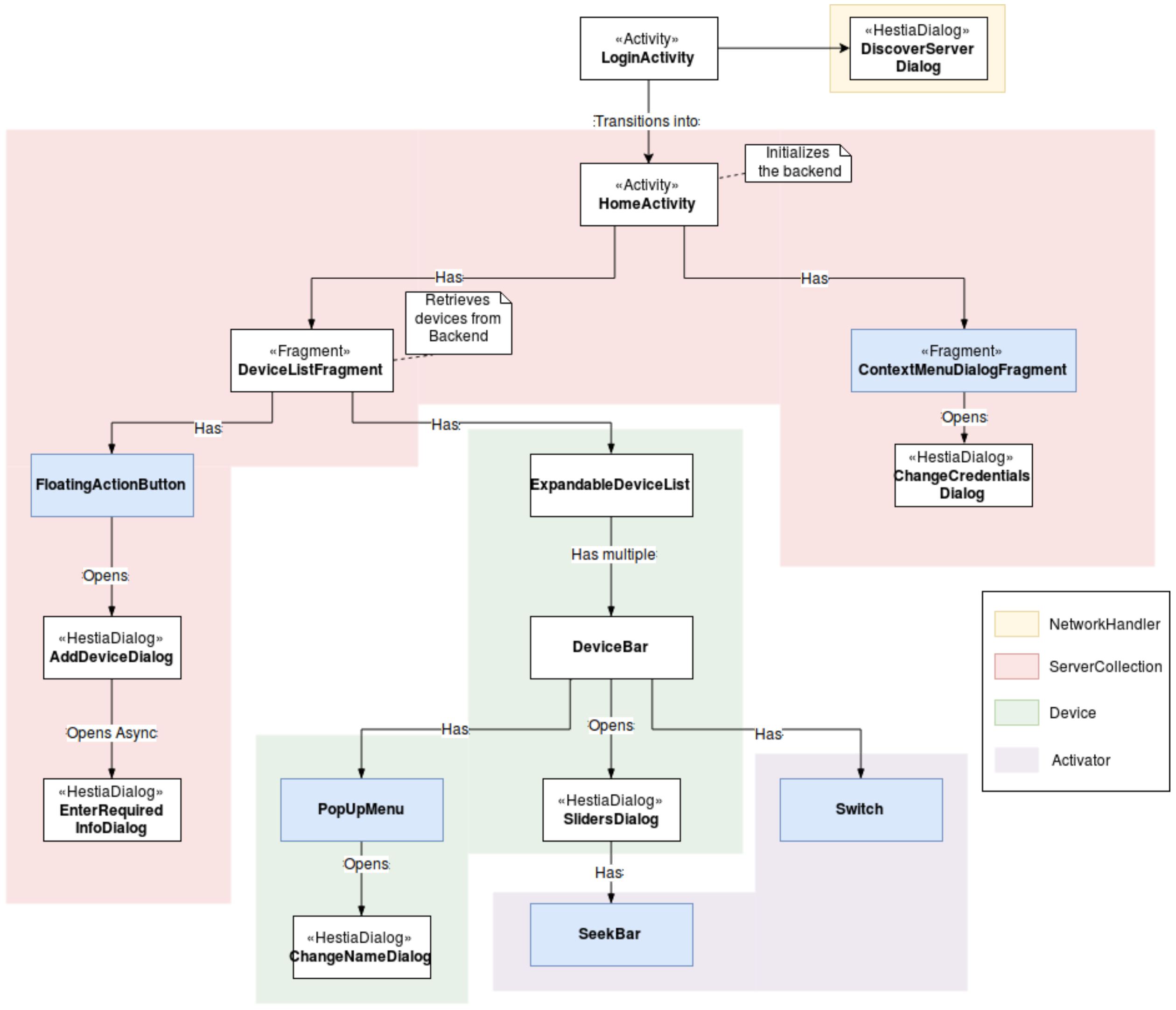
Within the DeviceListFragment there are multiple DeviceBar objects created for each device in the system. This DeviceBar is the main interaction point between the device and the user, which contains a variety of ways that can be used to interact with the devices. The different device bars are displayed in the DeviceListFragment using the ExpandableDeviceList. This class groups the different device bars based on the type of each device.

Whenever input of the user is required within the application, a dialog is used. Most of these dialogs inherit from the HestiaDialog. It is often within these dialogs that interaction with the backend occurs. As stated before we use the android native AsyncTask for this. The OnProgressUpdate methods of the ATs make it possible to simply catch the exceptions thrown by the backend and show a Toast message for them, or do something different when needed.

The diagram below shows the different elements of the graphical user interface. The blue boxes in the diagram represent elements that have their behavior defined in the class which contains them. For instance, the fact that the FloatingActionButton opens a new AddDeviceDialog is defined in the DeviceListFragment by setting the *onClick* listeners for the FloatingActionButton. Moving these declaration into the classes themselves will be explored in later stages of development.

The three different background colors indicate with what part of the backend the classes interact with.

Something that is missing from all diagrams are the xml definitions used for defining the layout of the GUI elements.


«Activity» LoginActivity
«HestiaDialog» DiscoverServer Dialog
Transitions into
Initializes the backend
«Activity» HomeActivity
Has
Has
Retrieves devices from Backend
«Fragment» DeviceListFragment
«Fragment» ContextMenuDialogFragment
Opens
«HestiaDialog» ChangeCredentials Dialog
Has
Has
FloatingActionButton
ExpandableDeviceList
Opens
Has multiple
«HestiaDialog» AddDeviceDialog
DeviceBar
Opens Async
Has
Opens
Has
«HestiaDialog» EnterRequired InfoDialog
PopUpMenu
«HestiaDialog» SlidersDialog
Switch
Opens
Has
«HestiaDialog» ChangeNameDialog
SeekBar
NetworkHandler
ServerCollection
Device
Activator

Technology Stack

The Android application is designed in Java using the available libraries from Android SDK 25 and Google's GSON library for creating and sending JSON objects. We decided to use JSON instead of other methods of marshaling data because JSON is very simple and easy to read for humans. It might be less expressive than XML, but it is powerful enough for our purposes. The readability of the JSON objects is also very useful for other people who might at a later point start using our code. The GSON library acts as a framework for serializing and deserializing JSON objects. A JSON object is mainly a HashMap whose information is in the (Key, Value) form.

Apart from this, the program uses the HttpsUrlConnection system[2] to handle all the requests over a HTTPS connection. To communicate with the Server, the Android application first needs to know the Server's IP address and port number.

Hestia Application also features automatic server discovery. This is achieved with the help of the Zeroconf framework[3]. This feature searches through the local network for the Server, which is listed as protocol "_hestia._tcp.". It was implemented using Android's NSDManager, which provides methods for implementing discovery and resolution listeners. The discovery listener searches through the Network, gathering all server's with the specified protocol. The resolution listener takes the discovered server and "resolves" them, i.e. give their IP address and Port number.

The Client side of the Hestia project was done using the Android Studio 2.3.1 (2017) IDE.

Design patterns

To facilitate easy communication between the frontend and the backend of the client, a facade pattern was used. The ServerCollectionsInteractor (SCI) class handles all signals from the frontend, thus making the relatively complex backend system invisible to the frontend code. The predecessors of what eventually became the ServerCollectionsInteractor also implemented the Singleton pattern, but after later restructuring we only needed the SCI in the home activity screen. This allows us to simply pass it around, we don't need it to be a Singleton anymore.

Another pattern used in Hestia Application is the composite pattern, which means composing objects into tree structures to represent part-whole hierarchies. This is reflected in our system especially when showing the list of devices. Each group of devices contain a list of devices of the same type. Each device contains an id, a name, a type, an instance of NetworkHandler and a list of activators. Each activator contains an id, a rank, a name, the device, an instance of the NetworkHandler and the current state. The state of an activator consists of a type and a raw

[2] The documentation for the HttpsConnection:
https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html
[3] Zeroconf wiki page: https://en.wikipedia.org/wiki/Zero-configuration_networking

state. The type of the activator will be used in order to properly parse the raw state. If the type indicates that the raw state is a Boolean, then the raw state could be either true or false. If the type indicates that the raw state is a Float, then the raw state could be any value in the range of a float. In other words, the type dictates the value of the state. This is an example of the Adapter pattern. The state, which can be of any primitive type, is wrapped in an ActivatorState class.

The Hestia client also uses the Front Controller pattern. This means that it handles all requests to a website, or in this case, a web service, namely the server. Whenever the user presses a button to add, delete or change that state of a device, an HTTP request is sent to the server. The client is thus a front for all interactions with the server using HTTP over a HTTPS connection.

Backend architectural decisions

In this section we discuss important decisions on the client side. The reason for choosing JSON over XML for communication is discussed at the server side part of this report, but it must be seen as a mutual decision from both client and server side.

Version of Android

Hestia application targets Android API level 16 and later, which corresponds to Android version 4.1 (JellyBean).

Reasons to use API level 16

- Targets a very large number of compatible Android devices.

Reasons to use any other API levels

- Lower API level would provide a better coverage, but it lacks in many features existing from API level 16 onwards.
- Higher API level would provide more features that could be used for this App, but it does not provide such good coverage.

Decision

By choosing this level, our application will run on at least 95.2% of all Android devices. We believe that this decision strikes a balance between having both coverage and the latest features. We ensure that this API level will not increase in the future, therefore users will not be dependent on the latest technology to run the application.

GSON versus Jackson

We considered two frameworks to use for JSON deserialization, Google’s GSON and Jackson. Both are fully capable open-source JSON libraries for composing and decomposing JSON objects

Reasons to use GSON

- Simplicity
- It is more Object Oriented than Jackson
- It is high-level

Reasons to use Jackson

- Custom creation of a JSON object
- Deserialization does not require a class to store information. But, if such a class exist, deserialization is not fully dependent on the contents of that class.
- Because deserialization is done manually, it is relatively low-level.

Decision

We went with GSON instead of Jackson for much the same reason as why we chose JSON over XML, namely simplicity. For simple use cases, GSON requires nearly no additional code, we just hand it an input and a class template and it will handle everything else. This kind of behavior was very useful for our project.

Frontend architectural decisions

Expandable List

We considered two possibilities of displaying the devices in the app, first of all the standard list of the devices, and next the expandable list view.

Visual design
A ‘main’ list consisting of sublists. Each list item from the ‘main’ list is clickable. Once clicked, a sublist, belonging to the list item, is expanded. This greatly increases usability, specially if there are more than 10 devices connected. In this way, the user does not have to scroll through all the devices, but only through the ones that are in the specified group. The difference between the expandable list and a standard list is that the standard list is a simple list, whereas the expandable list can collapse or expand a group of devices, which ultimately increases the user experience.

Architectural design
Concerning the architecture, a normal list without the option of being expanded or any options would be really easily implemented. This would just be the standard android list, which should be dynamic of course. On the other hand the expandable list is harder to implement. The expandableListView is also an android class which extends the ‘standard’ listView class. The only extra things we should add for this are ‘groups’ or types of devices which is therefore also an extra choice in the implementation of devices at the server.

Reasons to use standard list

- See all your devices at once
- Less steps to turn on/off devices
- Easier implementation

Reasons to use expandable list

- Clear overview of the types of the devices
- Ability to sort the devices in groups
- No extremely long lists when opening the app

Decision:
We decided to go with the expandable list, because this clearly gives a better user experience. As the user can easily see all devices of a certain type at once, and can close it once done. The extra step didn’t seem to really outweigh the advantages.

Login

We decided to use the “mock” login screen in combination with storing the hashed password and username because using actual security and encryption of communication was not feasible. The security decisions are described in more detail in its own section. We decided to use the login screen anyways for some added local security, so that not anyone using someone’s phone can access the entire application.

Hashing

We decided to hash both the password and username, because otherwise this will be stored on the phone’s memory in plaintext, which is really insecure. We use SHA-512 encryption because this is one of the best standard hashing algorithms in the Java MessageDigest library. We could further improve the security of a user’s credentials by using multiple hashing algorithms, but this process would also make the app slower. Therefore, SHA-512 suffices for our purposes.

Remember me

When the remember me button is ticked when logging in, a boolean called *savelogin* is set to true in the SharedPreferences. When someone opens the app, the LoginActivity always first checks if the *savelogin* boolean is set to true in the SharedPreferences, if this is the case, the user will never see the login screen and will be redirected automatically to the HomeActivity.

Change username/password

For increased usability and safety we added a change username/password dialog so the user can chooses his/her own username and password which can be used to login to the app. To edit either the username or password the old password has to be entered and has to be correct, for safety purposes. Also when editing your password, the new password has to be entered twice for confirmation.

Network Discovery Service

Hestia Application features automatic server discovery, using the Zeroconf framework. Pressing the button “Server” in the login screen, will open the DiscoverServerDialog and will trigger this process When the server is found, a message is displayed on the top of the dialog, including the IP address of the Server. Then, the user the can press the button “Autocomplete!” to automatically enter the IP address in the IP field. Afterwards, the user confirms the changes. If the server is not found, a message will be displayed and the user still has the possibility of manually entering the IP address.

Device bar

We decided to create device bars inside the expandable list. (each device has its own device bar) We had to make the choice between putting only the switch (on/off) button in the bar or also putting in the slider if it has one.

Visual design

From a visual point of view, there were a few options. First of all putting all the elements in the device bar itself is very crowded and very ugly as a consequence. Only putting in the on/off button makes sure there is improved usability as the user can easily access this one function, and this option doesn't look bad. Putting in none of the items looks clean, but really decreases usability as there are a lot more clicks for simple actions.

Architectural design

These three choices don't differ a lot in implementation. However, if we add the slider to the device bar, we must be able to properly size it in the small bar. If we were to remove all functions we must add extra steps, like for instance dialogs, in order to ensure the actions are still available, which will make the UI more complex.

Reasons to add nothing in the device bar

- Very clean design
- More space for info or description about the device

Reasons to only add switch

- Less crowded
- Looks a lot better for the user
- It is the primary action, the only one that needs to be quickly available

Reasons to add sliders

- Less clicks to access the sliders
- Easily see if a device contains a slider

Decision:

We decided to go with only the switch (on/off) button in the device bar. This decision was made mostly because a slider in the device bar would make it look worse, and the slider would be that small that it would be hard to use. So only the switch was the one to go with.

Dialogs

To get feedback from the user the graphical interface makes extensive use of the android native Dialog class. When shown these dialogs popup in front of the user interface. Depending on the what is requested from the user the dialog features different ways for the user to enter information. Each moment a user needs to add information there is a different Dialog class defined.

The decision we faced is how the base class for these dialogs should be defined. We wanted a HestiaDialog class for this, but since we are dependent on android native classes the HestiaDialog should either extend the simple Dialog class or the AlertDialog.

Visual design
The base Dialog class of android is very simplistic and allows to customise the design of the dialog in detail. The AlertDialog has a way stricter design. The customization options of the base Dialog make it however very hard to scale the design for different screen sizes.

An additional benefit of the AlertDialog is that it changes according to the android version the phone/tablet has. This makes the design of the hestia app more in line with what the user expects.

Architectural design
The AlertDialog has stricter rules in how the dialog should be build, it uses a special builder class for this. These stricter rules make it harder to create a solid HestiaDialog that the other dialogs can inherit from.

Reasons to use the base Dialog class
- Possible to make the dialogs look exactly as we want them to
- Easier to create the abstract HestiaDialog class

Reasons to use AlertDialog
- Dialogs scale better
- In line with standard android look

Decision:
We decided to go for the AlertDialog structure. This decision was based on the scalability of the dialog which would allow for a bigger group of possible users.

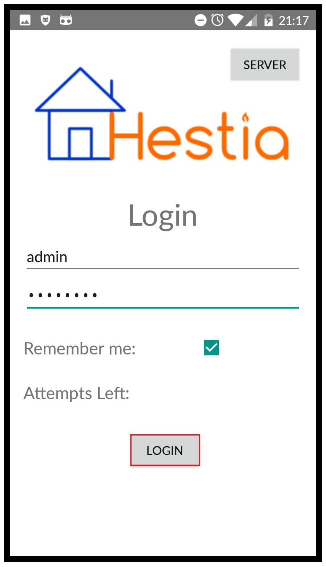
In the end we also decided that having a look in line with the user's expectations was also more important that having our own style.

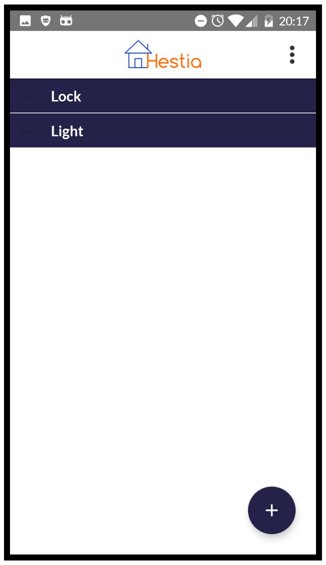
Graphical User Interface

User interactions

Logging in

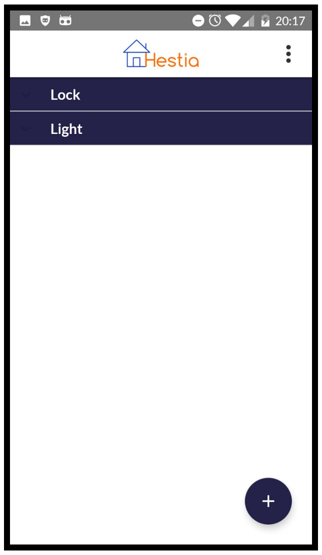
When opening the app, the loginActivity is opened as starting screen. The login screen has a username and password field, a remember me button, and a login button. Furthermore the screen contains the text “Attempts Left:” this indicates how many of your 10 attempts are left. After 10 attempts it is not possible to log in (restarting the app will also restart the counter). The textfields are pretty straightforward but when someone decides to check the “remember me” box, the application saves it in the shared preferences of your own device, this way one never has to login manually anymore. To achieve this, when the app opens the loginActivity, this activity first checks whether the shared preferences are set, and if this is the case, directly redirects to the next screen (HomeActivity).

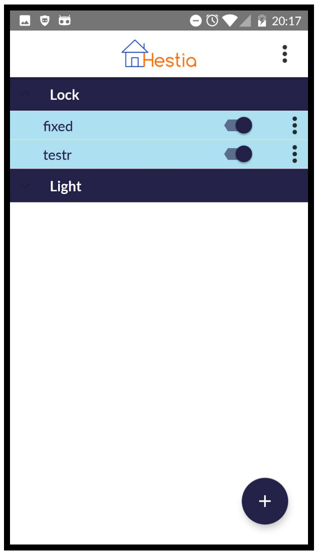




Opening a group

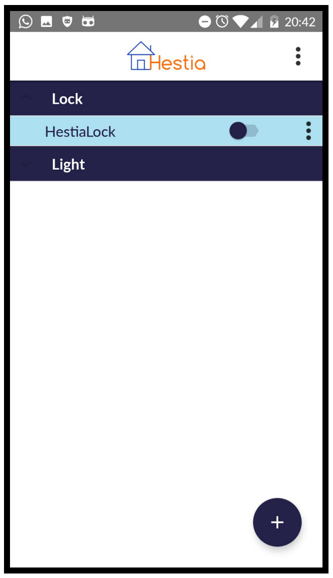
When opening a group item (for instance ‘Light’), all its children will be expanded. This is done by calling the method getChildView(), which is called for every child in the group.

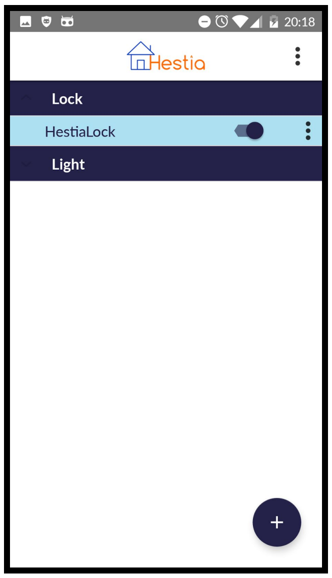




Turning on a device

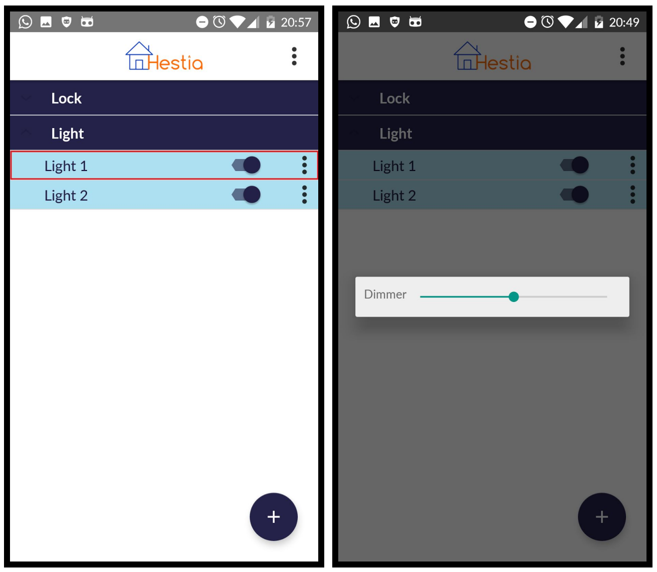
When pressing on a switch button, a post request is sent to the server. The server will then handle the corresponding message (i.e. turn device on or off).





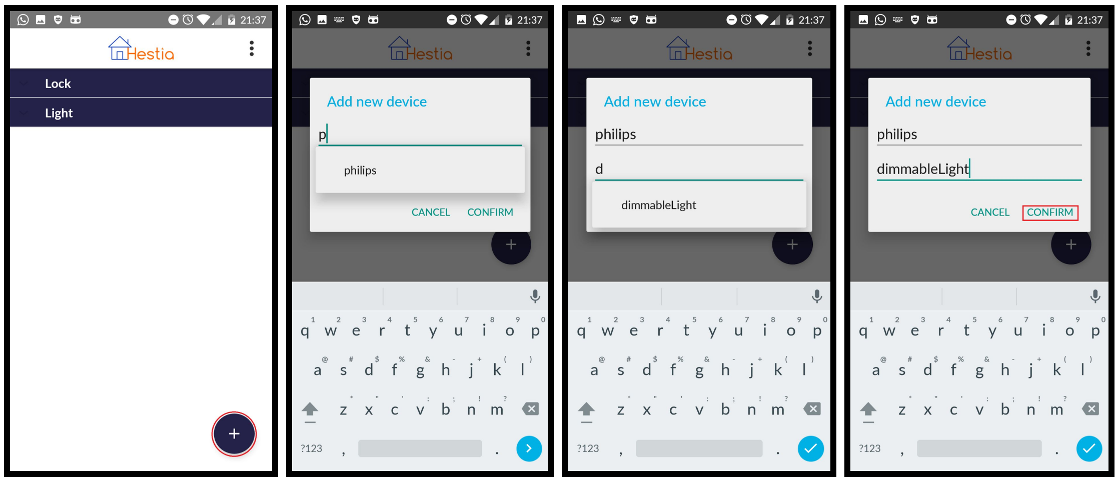
Slider

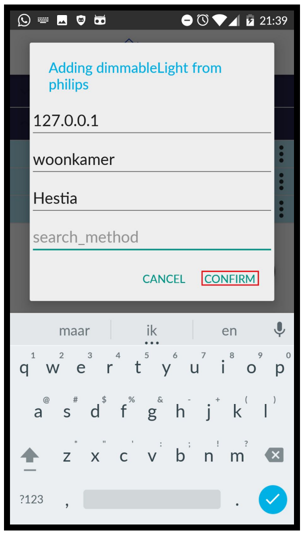
When clicking on a certain Device, a slider diagram will open (if the device has a slider). This slider can then be used to change, for instance, the brightness of a light.



Adding a device

When clicking on the button with the plus inside, a dialog is opened, asking for the Collection and the name of the Plugin. If the client is connected to a server, those fields show suggestions based on input. Once those fields are filled in and the user pressers confirm, a new dialog opens, letting the user fill in more fields. Then the backend will send this info to the server and the device will be added to the server. Finally, the newest list will be retrieved and will be shown in the GUI.





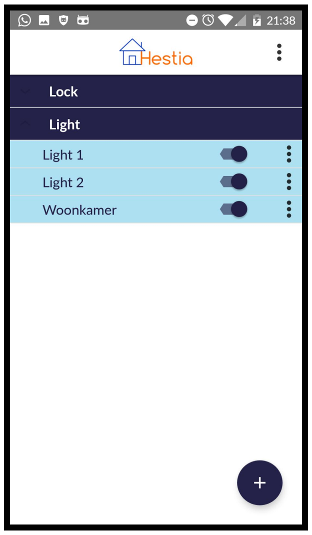
Adding dimmableLight from philips

127.0.0.1

woonkamer

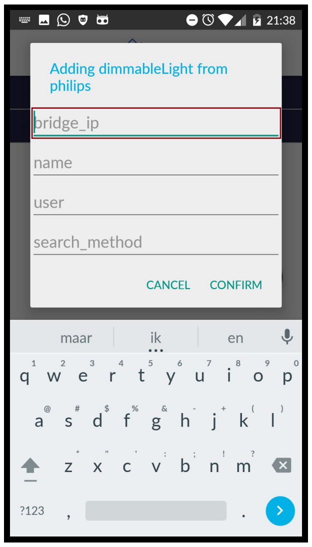
Hestia

search_method



Long press

When the user presses for a longer time on a certain input, a new dialog will be opened with information about that certain input.



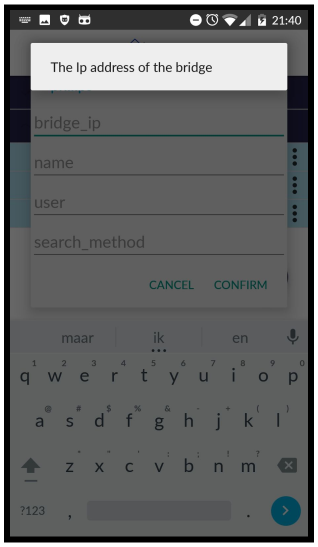
Adding dimmableLight from philips

bridge_ip

name

user

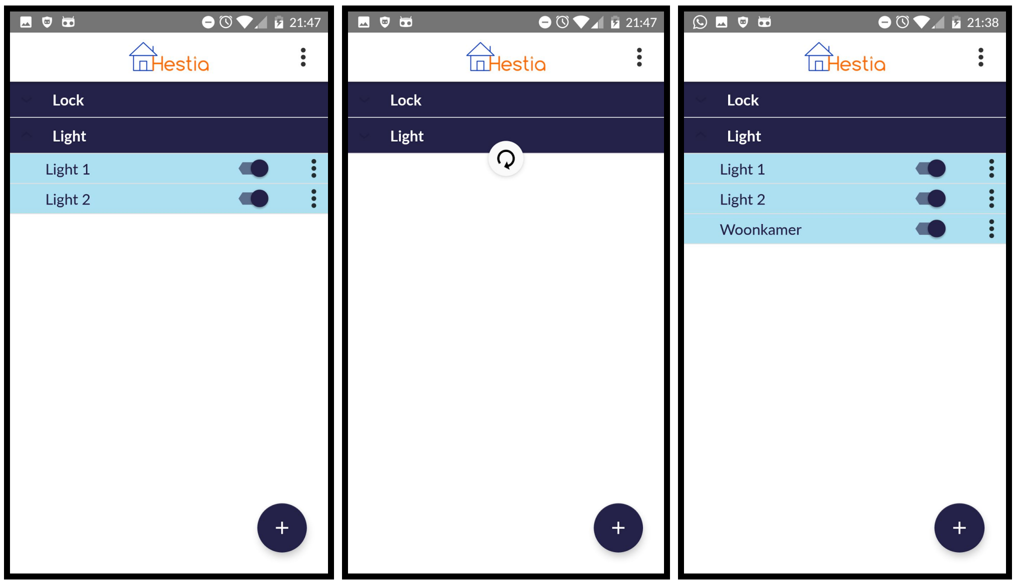
search_method



The Ip address of the bridge

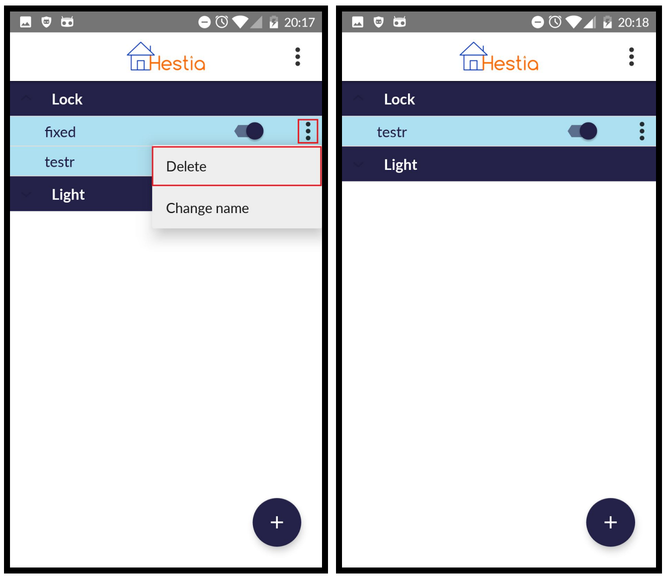
Refreshing

When refreshing the list. An asynctask is created to retrieve the latest list from the server. This is done by the backend. The backend will then return the list to the GUI. The list will then be sent to the ExpandableList which renders the new list.



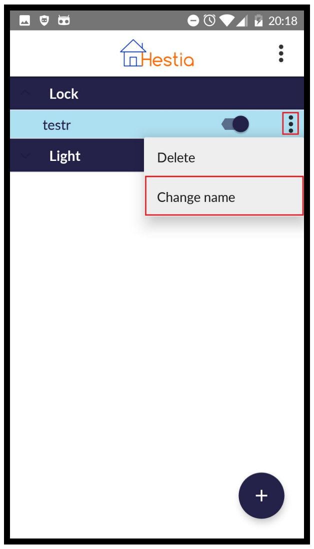
Removing a device

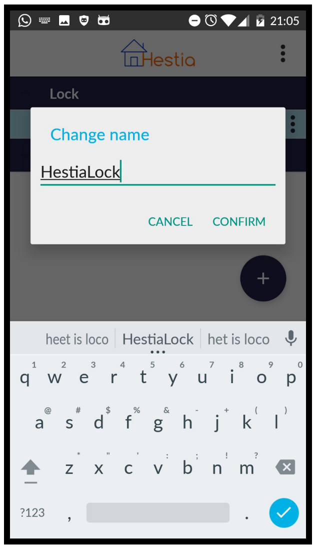
Pressing delete from the submenu of a device, the backend sends a delete request to the server. The server will process this request. After deleting, a get request is sent by the backend to the server to retrieve the newest list. This list will then be shown in the GUI.

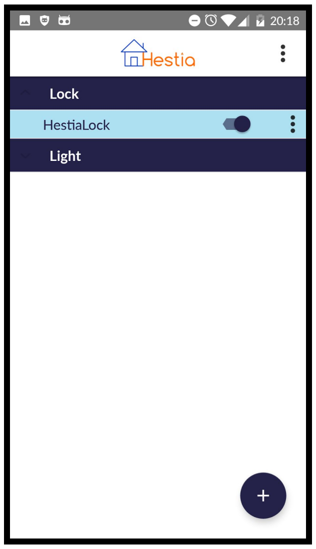


Changing the name of a device

When the user presses Change name, a dialog will pop up, letting the user changing the name of the selected device. This name will then be saved on the server and sent to the GUI by the back end part.

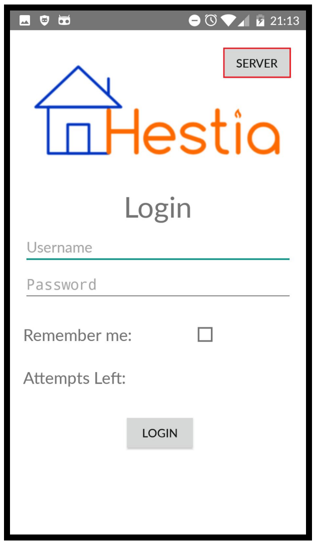


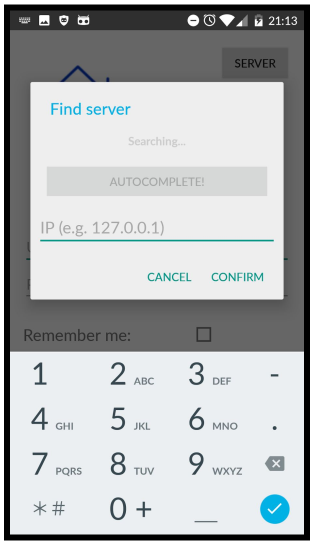




Network discovery

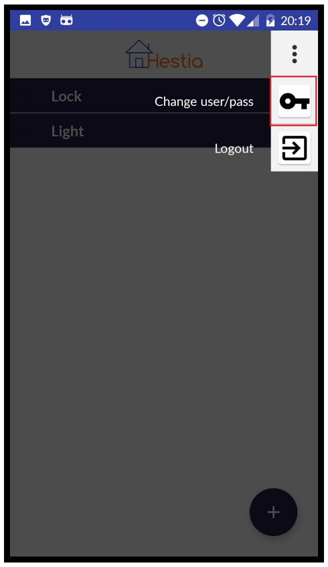
When pressing on the button 'SERVER', a dialog opens. In this dialog it is possible to either wait for the client to find the ip address of the server automatically or give the ip address as input. Once the user pressed confirm, the IP-address where the phone sends it requests to is changed to the new IP-address.

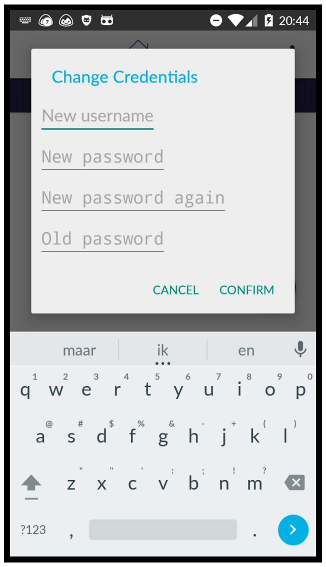




Changing credentials

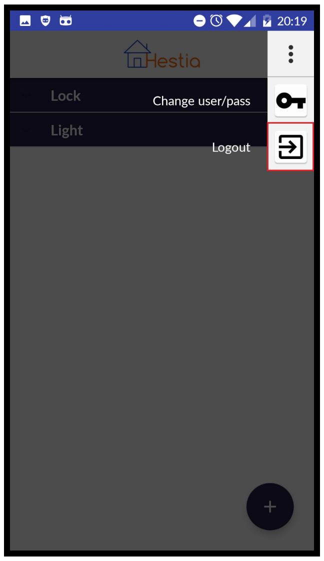
When pressing on the menu item ‘Change user/pass’, a dialog opens, asking for the new credentials. Once the user pressed confirm, the new credentials are stored in a hash.

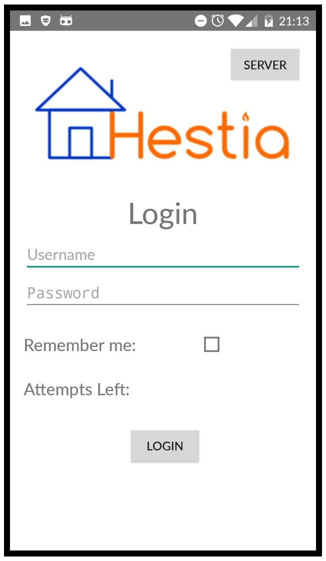




Logging out

When clicking on the options menu in the top right corner, the logout button appears. When the user clicks this button, the user is directly redirected to the login screen again and, of course, the sharedpreferences from the “remember me” are deleted so that the user won’t be automatically redirected again.

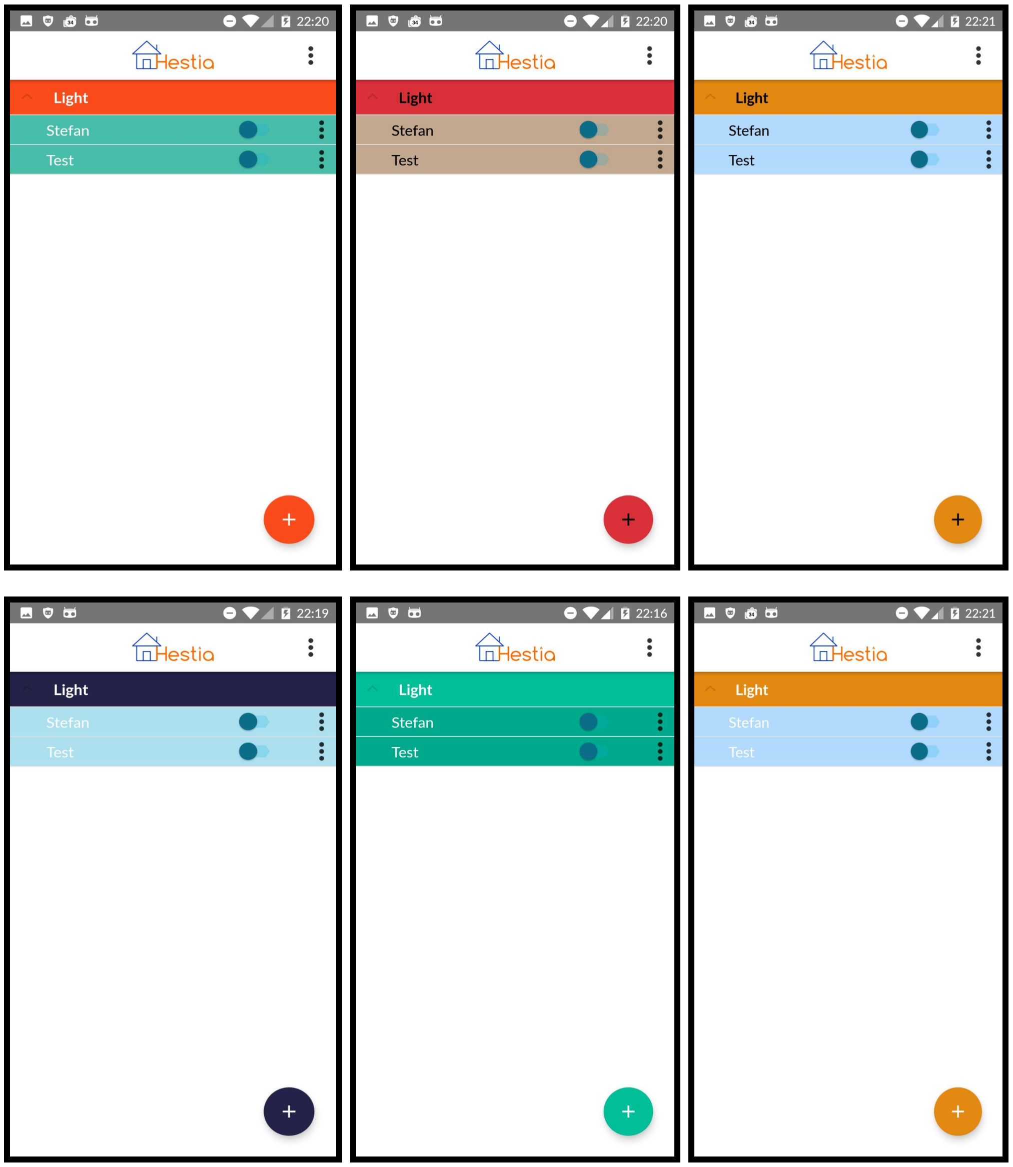




Color scheme

We chose the color blue as it represents professionalism, serious mindedness, integrity, sincerity and calmness. The combination of dark blue and light blue is used to make a clear distinction between a group and its children. The white background is used as it corresponds nicely with the colors dark blue and light blue. We furthermore tried to achieve good contrast in order for the names and other text to be very readable.

Alternatives:



Server

Architectural overview

The upper part of figure 3 displays all the resources, or endpoints, the server has at its disposal. These resources depict the endpoints that the client can use. The arrows between these endpoints show the relative uri parts that can be taken to reach each resource. When an outside source interacts with these endpoints the endpoints contact the logic layer of our server.

Within the logic layer the methods interact with the Database and the PluginManager, although not all logic classes use both of them. Using the Database and the PluginManager the logic layer can instantiate and access the devices.

By using the PluginManager the logic layer has access to the plugins the server offers. A plugin is an abstraction of a real world peripheral. Plugins define the actions a peripheral performs, this is done in the form of activators. Plugins contain several activators, each activator has a perform method. This method is used to perform the action on the peripheral, in most cases some kind of REST api call on the peripheral is used for this.

In the diagram the plugins can be seen as an implemented Device class with the implemented Activator classes attached to it. These implementations contain some class methods to request information about the peripheral. To use the methods to perform actions of the peripheral, the plugin needs to be instantiated. A full list of all the peripherals can be found in the appendix.

It is the job of the PluginManager to instantiate these plugins. For proper instantiation of the plugin it is given the information needed to communicate with the peripheral, which information this is, is defined in the required_info of a plugin. To distinguish between the uninstantiated plugins and their instantiated counterpart we call *plugins* devices after instantiation. The instantiation process of a device is finished by calling the setup method. This method's goal is to establish the first contact with the peripheral.

Once the PluginManager has instantiated a plugin into a device, this device can be added to the database by a resource. From now on other resources can use this device to communicate with the peripheral. To perform an action of the peripheral, the plugin layer can get the device from the database, set the state and call the perform action.

Besides the perform method the Activator class has, the Activator and Device classes are nothing more than a shell for the database. All methods to set information on the devices or activators directly add these changes to the database. When the developer of a new plugin wants to alter the behaviour of a device it can override one of the methods.

More information on how plugins should be structured can be found in the section dedicated for this. This section also contains a short description of how new devices can be created.

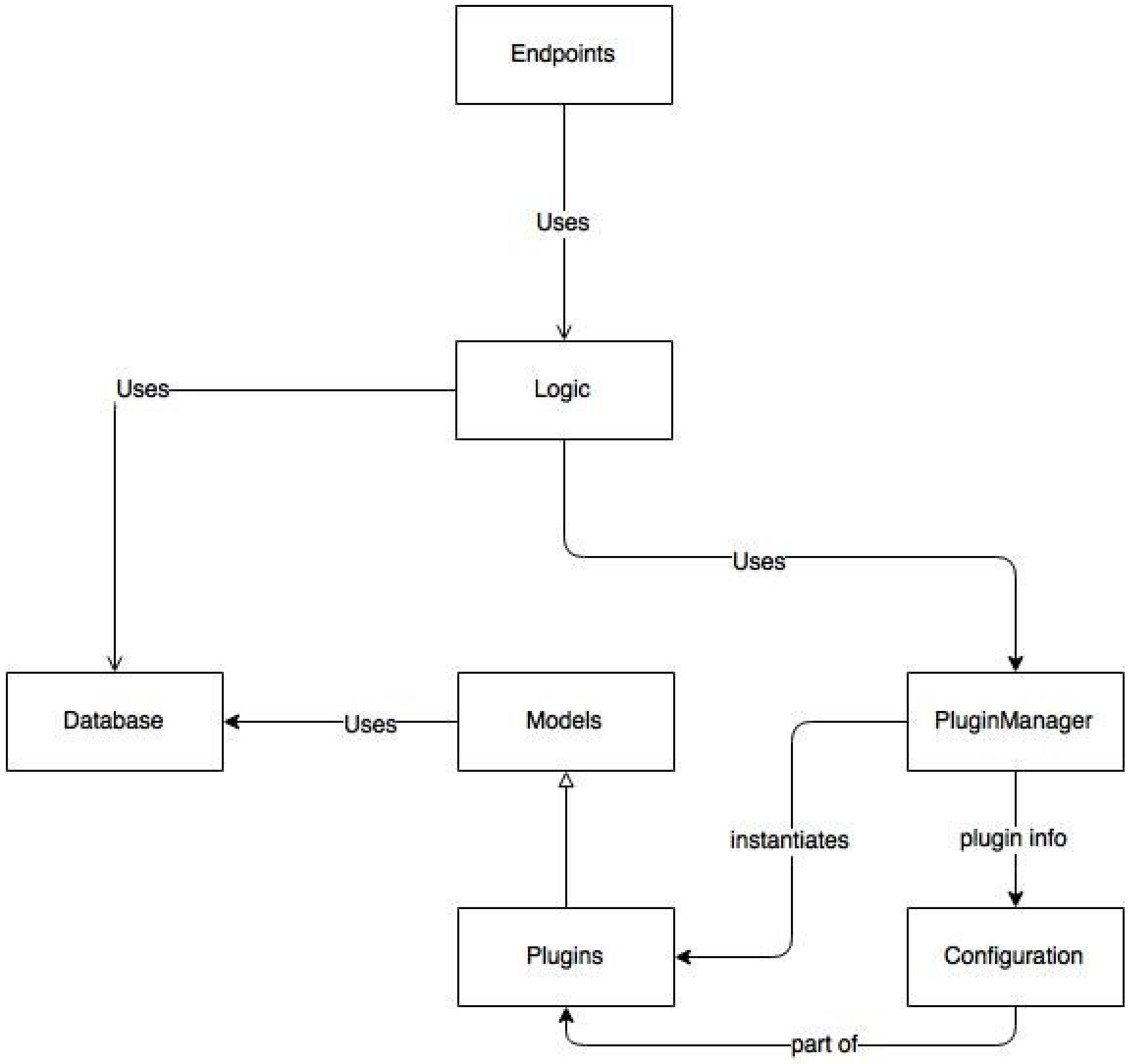


Figure 2: Abstract overview of architecture

Figure 2 shows an abstract overview on how different parts of the server work together and depend on each other. Figure 3 shows the abstract class diagram of the server where methods and variables are left out. All Resource classes have HTTP methods like GET, POST, PUT and DELETE. The logic behind the Resource classes is put into the Logic classes. The DeviceDatabase contains methods to retrieve, add or delete devices from the MongoDB database. The PluginManager instantiates plugins and is used in most of the Logic classes.

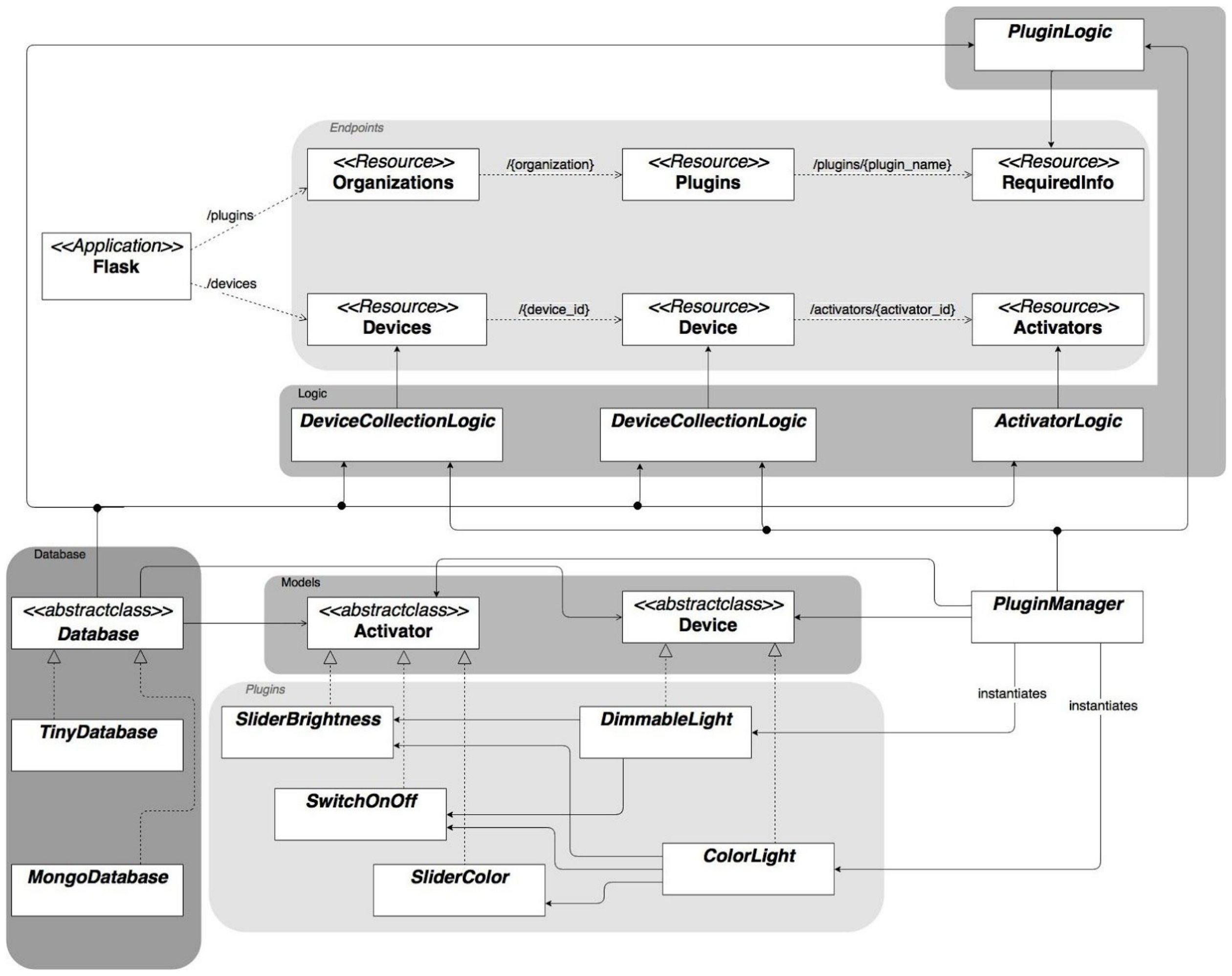


Figure 3: abstract class diagram

Technology stack

The server will be written in Python and will make use of the flask_restplus framework[4]. We refer the interested reader to the section on important architectural decision. Here we explain the reasons for using a REST api for the server and why we use python in combination with the flask_restplus. In the server we will separate the endpoint (REST api) from the business logic and the plugins. For the database we will be using tinyDB[5].

For the discovery feature we use the Zeroconf. Zeroconf refers to a grouping of several protocols that are combined to create an easily discovered IP network without special configuration servers. The server uses the python-zeroconf[6] library for this. At the time of development this was the only library available with pip.

[4] https://flask-restplus.readthedocs.io
[5] http://tinydb.readthedocs.io
[6] https://github.com/jstasiak/python-zeroconf

Plugin structure

For a description of the plugin server we refer to the ReadMe[7] of the development branch on GitHub.

Important architectural decisions

Below, we will elaborate on some specific major architectural decisions and the reasons we made a specific choice.

Transition to the use of third party devices instead of developing devices our self.

Reasons for creating our own devices:

- Easy integration with the Hestia server
- We could allow the Hestia platform to perform actions that are unique for our platform

Reasons for using third party devices

- Users can switch to the Hestia platform without needing to invest in new devices
- A wide variety of devices are already on the market, creating them ourselves would require more time then implementing existing devices in the Hestia platform.
- Development of the enclosure of the devices would require a third party engineer
- Lot of overhead knowledge required for developing our own devices, examples are networking, power consumption, safety regulations, etc.

Decision:

After a meeting with the customer, the decision was made to move to third party devices. This decision was mainly made because we do not have the knowledge at hand to handle the complete development of a device.

To keep the project in line with the customer's original needs and to distinguish ourselves from other platforms for home automation we will continue the development of the lock. It will however be transformed into a more general purpose device. The device will be stripped down to just the motor, the user can then use this motor to create his own devices; a lock for instance.

[7] https://github.com/RUGSoftEng/2017-Hestia-Server/tree/master

Protocol for communicating between the phone and the server

Reasons for using our own socket level protocol

- No time investment in learning an existing protocol
- A big part of the server is based on our own protocol this will need to be rewritten when switching to an existing protocol
- No restrictions on the kind of messages the protocol can handle since we define it ourselves

Reasons for using REST

- Lots of documentation readily available
- No big structural changes needed in our software when adding new functionalities
- Allows for a wide variety of clients; phones, website, etc.
- It will make it easier for the client to continue the development after our course is done

Other considerations

- SOAP : Way more knowledge required then REST, this will make the development process slower

Decision:

We decided to take our losses early in the development process of the platform and start from scratch in our second sprint and this time use a REST framework . The client was made aware of this decision and the resulting pause in the progress of the platform. During a meeting with the client it was made clear to us that they are happy with the transition and the fact that we took into consideration the problems that might arise when the project is handed back to them.

Which framework do we use to create a REST web service

Reasons for using Dropwizard/Java[8]

- Dropwizard can deal with a high number of requests per second.
- We are all familiar with Java
- Java is strongly typed, this might prove useful later.

Reasons for using Flask_RestPlus/Python[9]

- Easy to set up
- Creates documentation on the fly
- Will probably cause less problems when we want to use different value types for controlling the devices
- Client is familiar with the framework

Decision

We made a sample web service in both frameworks and compared which framework we felt most confident about. We initially had some disagreements among the team members. After weighing the pros and cons we decided to go with the Flask framework.
Moving from Java to Python will cost some time, since not all of us have experience with this. However, the simplicity of python should allow quicker development in the future.

Json or XML

Reasons for using Json

- Very readable, no duplicate text.
- Preferred by customer.
- Object oriented and therefore easy to import data in Python and Javascript.

Reasons for using XML

- Extensible, any data type can be stored in XML.
- Complete integration of all formats.
- Very readable.

Decision

When we discussed these choices with the client, they said they would preferred JSON, so that is what was chosen.This is in agreement with our customer.

[8] http://www.dropwizard.io/1.1.0/docs/
[9] https://flask-restplus.readthedocs.io/en/stable/

Database for device persistency during restart

Reasons to use a NoSQL database, specifically mongoDB[10] & TInyDB[11]

- High flexibility for storing the information needed to communicate with the peripherals.
- Easy to work with and change during the development phase
- Customer is familiar with it

Reasons to use a SQL database

- Most of the team is familiar with it
- Restrictions it can impose might eventually be useful for limiting the plugins developers possibilities.

Decision

It was decided to use MongoDB & TInyDB for our database. The main reason for choosing a NoSQL database is that these databases are significantly more flexible than SQL database, making it easier to store additional information for devices and plugins. MongoDB was also referenced during the lectures by our customer. However MongoDB is somewhat difficult to set up. It requires to install a MongoDB server and configure it correctly , which both are platform dependent. Therefore, we choose to add an additional database in the form of TinyDB this is a completely database implemented in a python library, using an on disk file. With using tinyDB as default database we allow for easy set-up of our system independent of platform. However, for deployment in an environment where a lot of devices will be added to the system switching to the MongoDB database will likely result in a faster system.

There are not direct references to the either of the databases in the code. Everything is wrapped in the Database class. In the case that we want to switch database systems this can be done without changing any code outside this class. (Tests excluded)

[10] https://www.mongodb.com/

[11] http://tinydb.readthedocs.io/en/latest/

Error handling at the server side

There is a variety of things that can go wrong at the server side when the client does a request using the REST api. In the case that something goes wrong the server throws an exception. In some cases the server wants to handle these exceptions itself, but in other cases the client should be notified of this issue. To handle this the server transforms the exceptions into a json object that the client can parse.

The REST framework we use, Flask, has a decorator to do this transformation. However the version we use only supports the transformation from an exception to a HTML format, while we require a json format. Therefore we need to catch the pre-existing exceptions in the server code and translate the exceptions to the json format ourself. Once translated we can use the flask abort method to pass the json format to the client.

Reasons for catching the exceptions in the logic module

- Keeps the endpoints clean and concise.
- Most of the exceptions have to do with the logic, therefore it is a logical place to handle it.

Reasons for catching the exceptions in the endpoints

- While there is still a lot of code duplication it would be even more in the logic module
- When catched the exception is translated to json, this fits into the endpoint since this is the only module that should deal with json
- Exceptions thrown in calls to the logic module are still caught.

Decision:

We chose to put the exception catching in the endpoints. Our main consideration with placing it here was that we wanted to keep the endpoints clean. We decided that it was not worth it to make an already complex part of our system even more complex just to keep one module clean.

Security

Currently the Hestia server does not require the user of the API to be authenticated. Due to this we have no way of identifying who is performing the API calls. This means that everyone who has access to the network on which the server operates also has access to all the peripherals operated by said server.

To optimize the security of our platform there are two steps that need to be taken. First of all the connection between the server and the client needs to be secured. Secondly, the server should implement some form of user management to handle authentication. During the last iteration we had a look at possible solution for these problems and implemented what we deemed feasible.

In this section we discuss a few recommendations we have to the customer for further implementation of the security. Besides this we would also like to add a few warnings for the potential users.

Recommendations to customer

Encryption between server and client

In version 0.7 of the server we updated from HTTP to HTTPS as a communication protocol between the client and server. This encrypts the communication between the client and the server. To connect to a HTTPS server, a server certificate is sent to the client. The client can then check the certificate against its own library of known and verified certificates. If we want the server certificate to be accepted by the client, we need to obtain a certificate from a known Certificate Authority. In our current setup, this is not possible, as we need at least a Domain Name in order to be eligible for receiving a signed certificate. We cannot distribute a Domain Name to each server, and therefore it is not possible to get a signed certificate.

Another way to tackle this problem is by writing our own certificate verification system on the Client, which would only accept a Hestia Certificate. This proved to be very complicated on the Client side, and we decided not to implement it. This would still mean that each server would need its own certificate, which leaves us with the problem of distributing the certificates to the servers. In consultation with the customer the choice was made to accept all certificates on the client. This means that although the communication is more secure as with a default HTTP connection the communication is still vulnerable to a man-in-the-middle attack.

In the future, the system could be secured in a number of ways. We might allow users to add their own certificates, but this would mean true security could only be achieved by real power users. Another possibility is to show the certificate as a QR-code, but this would require some sort of display for the Hestia Server. We could also register a single domain, on which we can run a web service which can hand out certificates to Hestia servers. Finally, we could sell the system together with a key, which we could then preconfigure. This only works if we completely sell and control the server product.

User management and authentication

The Android application currently has a login screen. The credentials entered in the login screen are checked against values stored on the device. In the future, the app should send these credentials to the server over a secure connection. The server can then return a token over the same secure connection which would start a session. The username and password would be stored on the server. This is not currently implemented because of the amount of work it requires, but it would mean that the server is not accessible to people on the local network who don't have a username and password.

A second reason for not implementing user management is that we don’t have a solution for some issue for expiration of the token. Allowing tokens to be used indefinitely would essentially mean that there is no security at all. On the other hand using tokens that expire would mean that there is the possibility that the user has to log in at an inconvenient time. To overcome this issue we could store the password at the phone, but with our limited knowledge of how save the android storage is we don’t want to store plaintext passwords on the phone at the moment.

Additionally user management also means that it should be possible to create new user accounts for client server communication. This would require new endpoints for the server as well as multiple new dialogs on the phone, or perhaps a new activity.

Warning for users

Below we have compiled a short list of warnings and guidelines, based on the discussion above:

- The Hestia home automation system should only be used within a local, private network. Using it outside a private network (over the whole internet, for instance) is possible, but is very risky security-wise.
- Hestia should not be used to control critical peripherals, such as locks, to prevent home invasions from happening.
- The user should change the default username and password as soon as the Android application is installed.

Team Organization

We have decided to start using some Scrum practices in our team organization. During previous sprints we had a strong division between different groups working on different parts of the system. Getting this division out of our system and working closer together is our main point of focus for the team organization at the moment.

To facilitate in merging the different groups into one cross functional team we altered the way we defined requirements. During previous sprints requirements were defined for each component of the system. This allowed the teams to work on different requirements without needing to much interaction between the two teams.

We now use user stories to guide our development process. This way it is harder to work on the different components without knowing about the whole system. While this slows down development in the beginning it should allow us to work faster and cleaner after a few sprints. This is due to the fact that with this new structure knowledge about the system is spread more across the team. This should limit the time spend waiting on the work of other people and lower our “bus factor”.

Another advantage of working on user stories instead of the more concrete requirements is how user stories directly correlate to value for the user. This way each user story that is finished gives the feeling that the actual progress is made.

Scrum events

A big part of scrum are the events defined for each sprint, the stand up, the sprint planning, the sprint review and the retrospective. We try to do all these events in our sprints. However, they can take quite a while to perform fully so we made some small adjustments and shortened the time box of the planning, review and retrospective.

We perform the daily standup on a non daily basis. This is because it is too often that we can’t work on the project for a day. To keep forcing people to answer these questions would make it less useful. The sprint review, planning and retrospective are all done at the first Wednesday of each sprint at five o’clock. The sprint review is open to be visited by everyone involved with the project.

Jira

To help us with keeping track of the user stories we have selected to work on within a sprint we use JIRA. Every member of the team has access to our scrum board here.

Build process and deployment

Building is an essential part of the development process. After building, we can see whether our code compiles correctly.

Continuous Integration

We use Travis CI for continuous integration. This distributed CI service builds our code according to a configuration file (*.travis.yml*) located at the root of our project. We use it on both the server and the client.

For the client, Travis installs the Android SDK, along with other build tools, such as Gradle. It also downloads some system images to emulate different android devices. The build process consists of compiling the code, starting the emulator, running tests and then generating test coverage reports. On Github, a badge is displayed in the README file to indicate whether the build succeeded.

For the server the process is somewhat simpler, Travis installs all the requirements in the *requirements.txt* file and runs the tests (including coverage).

Test Coverage

For test coverage, the server and the client used different systems.

The client relies on Codecov, which uploads the coverage reports generated during the build process in Travis. The overall test coverage is displayed with a badge in the README. Specific reports can be viewed by clicking on the badge. On every pull request, a comment is placed with a coverage report.

The server uses coveralls for calculating the code coverage. After a successful test it will print the coverage report. The test will fail when the test coverage is too low.

Appendix

Glossary

Peripheral : The concrete object in the real world. Examples of these are the philips hue lights.
Plugin : The class containing all the methods and activators to communicate with the peripherals.
Device : An instance of an plugin that is added to the system and can be controlled through the client
Action : An action that is performed by the peripheral, for instance a light turning on or turning off.
Activator : Represents the action of the peripheral in the form of a state that can be changed. The action of a light turning on would for instance be represented by the activator going from the state False to True

Required info: This information is needed for the device to function. The user is requested to enter this information upon installation of the plugin. During the setup process of the device this information is used to create the options of a device.
Options : The options of a device are created during the setup process of an device. The options are used by the activators of the device to communicate with the peripherals.

RESTful API

The full documentation of our RESTful api can be found at http://localhost:8000/ when the server is running locally. We can separate the API into two different name spaces namely the devices namespace and the plugins namespace. Both will be explained below.

GET .../devices/

Returns a list of all devices currently installed on the server

POST .../devices/

Install a new device on the server
404: NotFoundException
400: SetupFailedException

GET .../devices/{int:deviceId}/

Returns the installed device with the given id
404: NotFoundException
404: DatabaseException

DELETE .../devices/{int:deviceId}/

Delete the installed device with the given id

GET .../devices/{int:device_id}/activator/{int:activator_id}

Returns based on the activator_id the activator on the device with the given divice_id.
404: NotFoundException
404: DatabaseException

POST .../devices/{int:device_id}/activator/{int:activator_id}

Changes the state of the activator on the device as specified in the URI to the state that is given in the body of the request.
404: NotFoundException
400: InvalidStateException
404: DatabaseException

GET .../plugins/

Returns a list of all collections that have a plugin currently installed on the server

GET .../plugins/{string:collection}/

Returns a list of all plugins of an collection currently installed on the server
404: NotFoundException

GET .../plugins/{string:collection}/plugins/{string:plugin_name}

Returns the required information for a plugin from an collection.
404: NotFoundException

Plugins

Collection : philips

colorLight

The philips color light can be used to communicate with all philips hue lights that contain the possibility to change color based on a scale of 0-65535. Upon instantiation the philips hue link button needs to be pressed.

Activators

SwitchOnOff(bool) : Turn the light on or off
SliderBrightness (float): Set the brightness of the light
SliderColor(float): Set the color of the light

Required info

bridge_ip : IP address of the philips hue bridge
user : username used to communicate with the hue bridge

- *“unknown” / “”* : creates a new user

search_method : Method that should be used to create the device

- *“last”* : looks for the device that was added the last
- *“reachable”* : this requires that only one light is reachable
- *“<name>”* : looks for the device with the given name in the bridge

dimmableLight

The philips Dimmable light can be used to communicate with all philips hue that are dimmable based on a scale of 0-255. Upon instantiation the philips hue link button needs to be pressed.

Activators

SwitchOnOff(bool) : Turn the light on or off
SliderBrightness (float): Set the brightness of the light

Required info

bridge_ip : IP address of the philips hue bridge
user : username used to communicate with the hue bridge

- *“unknown” / “”* : creates a new user

search_method : Method that should be used to create the device

- *“last”* : looks for the device that was added the last
- *“reachable”* : this requires that only one light is reachable
- *“<name>”* : looks for the device with the given name in the bridge

Collection : mock

light

Can be used as a mock device to test the functionality of a device with multiple activators.
Activators

LightSwitch(bool) : Mock activator that uses booleans
LightDimmer(float): Mock activator that uses integer values

Required info

ip : the device does not actually establishes a connection with this ip
port : same as above

lock

A device that was originally used to communicate with the in home developed lock. Can now be used as a mock device to test the functionality of a device with only one activator.
Activators

ActivateLock(bool) : Mock activator that uses booleans

Required info

ip : the device does not actually establishes a connection with this ip
port : same as above

Changelog

Who	When	Which section	What
S. Zijerveld	March 08 2017	The document	Created the document.
S. Zijerveld	March 08 2017	Team organization	Wrote down how we organize our group.
S. Evanghelides	March 08 2017	Brief Introduction	Briefly described the application.
S. Evanghelides	March 08 2017	Architectural Overview	Added information about the functioning of the Android Application.
S. Evanghelides	March 09 2017	Technology Stack	Added information about the design of the Android Application.
L. Holdijk	March 09 2017	Architectural overview and Technology stack	Added information concerning the peripherals.
L. Holdijk	March 21 2017	Important decisions	Added an additional section where we can describe our decisions more in depth. Also added 3 decisions to this section.
S. Zijerveld S. Maquelin	March 23 2017	Architectural Overview Technology Stack	Changed it to the rest implementation
S. Zijerveld S. Maquelin	March 23 2017	Whole document	Final touches before end of sprint 2
S. Evanghelides	April 13 2017	Client section	Updated all subsections of this section. Added class diagram
F. te Nijenhuis	April 13 2017	Client section	Updated general description and pattern sections
Lars Holdijk	May 01 2017	Team organization	Rewritten

Lars Holdijk	May 01 2017	Peripherals	Made changes regarding names
Suzanne Maquelin	May 01 2017	Server, Architectural overview	Changed class diagram and added explanation
Lars Holdijk	May 14 2017	Plugins	Changed activator names and indicated that some plugins are deprecated
Lars Holdijk	May 14 2017	Architectural overview, Important design decisions	Added information about the database MongoDB
Stefan Evanghelides	May 14 2017	Client: Architectural overview	Updated information about HTTP requests
Stefan Evanghelides	May 15 2017	Client: Architectural overview	Updated the whole section
Lars Holdijk	May 28 2017	Whole document	Changed all references to organizations to collections
Stefan Evanghelides	May 28 2017	Client: Architectural overview	Updated backend part. Reviewed frontend part
Lars Holdijk	May 28 2017	Client: Architectural overview	Added the new diagram for the backend
Lars Holdijk	May 28 2017	Client: Architectural overview	Update the description of the front end and added a new diagram
F. te Nijenhuis	May 28 2017	Client: Architectural overview	Removed outdated parts and added new text
F. te Nijenhuis	May 28 2017	Backend architectural decisions	Removed outdated parts discussing the usage of the Singleton pattern and created specific part for the UI decisions
Stefan Evanghelides	May 29 2017	Client	Small changes over the whole section
Lars Holdijk	May 29 2017	Frontend architectural decisions	Added decision between Dialog and AlertDialog

Lars Holdijk	May 29 2017	Glossary	Added required info and options
Lars Holdijk	June 6 2017	Restful API	Updated exceptions
Lars Holdijk	June 6 2017	Security	Added section
Lars Holdijk	June 9 2017	Server: Technology stack	Added Zeroconf description.
Lars Holdijk	June 9 2017	Error handling decision	Added
Stefan Evanghelides	June 11 2017	Architecture Overview, Frontend architectural decisions and Technology Stack	Added information about Network Discovery Service and zerocnof
Lars Holdijk	June 12 2017	Archtectural Overview client	Update diagram
Stefan Evanghelides	June 12 2017	Graphical User Interface	Aligned screen shots.